Deep Learning

Transformers: Da Atenção em RNNs à Arquitetura Completa

Lucas Silveira Kupssinskü

Machine Learning Theory and Applications Laboratory
PUCRS

agosto de 2026

Visão Geral

1. Atenção em RNNs
2. Scaled Dot Product Attention
3. Self-Attention
4. Positional Encoding
5. Arquitetura Transformer
6. Interpretabilidade

Visão Geral

O Problema do Gargalo (Encoder-Decoder)

Em arquiteturas *seq2seq* clássicas com RNNs, o **encoder** comprime toda a sequência de entrada em um único vetor de estado.

O gargalo

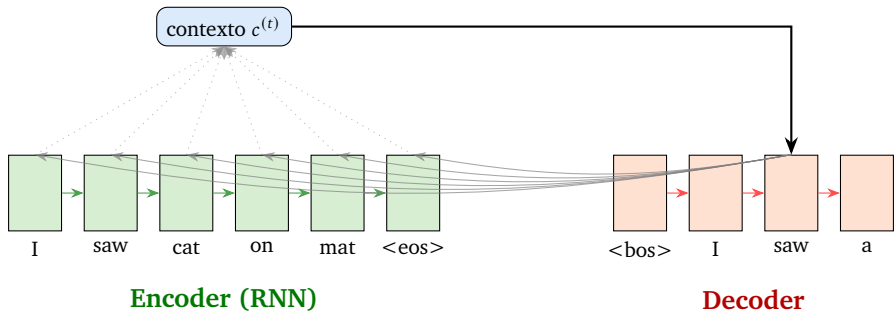
- Frases longas precisam caber em um vetor de tamanho fixo
- O **decoder** acessa apenas o último estado oculto do encoder
- Informação inicial da sequência tende a se perder

Solução: mecanismo de *atenção*

A **atenção** é um bloco da rede que conecta encoder e decoder e atribui *scores* a cada etapa do encoder. Interpretamos esses scores como uma decisão de quais partes da sequência de entrada são mais relevantes para gerar o próximo token.

Bahdanau et al., 2014. *Neural machine translation by jointly learning to align and translate.*

Encoder-Decoder com Atenção (Esquema)



A cada passo t do decoder calculamos um **contexto** $c^{(t)}$, uma soma ponderada dos estados do encoder, com pesos dados pela atenção.

Atenção: Scores, Pesos e Contexto

A cada passo t do decoder com estado h_t e estados do encoder $s_1, \ldots, s_m$:

Pipeline

1. **Score** entre o estado atual e cada estado da fonte:

$$\text{score}(h_t, s_k), \quad k = 1, 2, \ldots, m$$

2. **Pesos da atenção** (softmax sobre os scores):

$$a_k^{(t)} = \frac{\exp\big(\text{score}(h_t, s_k)\big)}{\sum_{i=1}^{m} \exp\big(\text{score}(h_t, s_i)\big)}$$

3. **Contexto** (soma ponderada dos estados da fonte):

$$c^{(t)} = \sum_{k=1}^{m} a_k^{(t)} \, s_k$$

O contexto $c^{(t)}$ é então combinado ao estado do decoder para produzir a próxima saída.

Como Calcular os Scores

Existem várias formas de definir $\text{score}(h_t, s_k)$:

Variante	**Fórmula**	**Referência**
Dot-product	$\text{score}(h_t, s_k) = h_t^\top s_k$	—
Bilinear	$\text{score}(h_t, s_k) = h_t^\top W s_k$	Luong et al., 2015
MLP (aditiva)	$\text{score}(h_t, s_k) = w_2^\top \tanh\big(W_1 [h_t; s_k]\big)$	Bahdanau et al., 2014

- **Dot-product:** mais simples, sem parâmetros adicionais
- **Bilinear:** adiciona uma matriz de pesos W aprendida
- **MLP:** mais expressiva, mas mais cara computacionalmente

A versão **dot-product** domina as arquiteturas modernas (transformers).

Visão Geral

1. Atenção em RNNs
2. **Scaled Dot Product Attention**
3. Self-Attention
4. Positional Encoding
5. Arquitetura Transformer
6. Interpretabilidade

Por que “scaled”?

Treinar redes com *dot product attention* pode causar **vanishing gradient** quando as dimensões dos vetores são grandes.

Intuição

Para vetores de dimensão d grande, $h_t^\top s_k$ tende a ter magnitudes grandes. A softmax sobre esses scores satura, fazendo com que quase todos os pesos da atenção fiquem próximos de 0 ou de 1.

Vamos quantificar: suponha $s_i, h_i \sim \mathcal{N}(0,1)$ i.i.d., componentes dos vetores $S, H \in \mathbb{R}^d$.

Calcule $\mathbb{E}[S \cdot H]$ e $\mathrm{var}[S \cdot H]$.

Esperança e Variância do Produto Escalar

Esperança:

$$\mathbb{E}[S \cdot H] = \mathbb{E}\left[\sum_{i=1}^{d} s_i h_i\right]$$
$$= \sum_{i=1}^{d} \mathbb{E}[s_i]\,\mathbb{E}[h_i] = 0$$

Para $s_i, h_i \sim \mathcal{N}(0,1)$ i.i.d., $\mathbb{E}[s_i^2] = \mathrm{var}(s_i) = 1$ (pois $\mathbb{E}[s_i] = 0$), o que justifica os dois últimos passos.

Conclusão: $\mathrm{var}[S \cdot H]$ cresce linearmente com d.

Variância: por independência,

$$\mathrm{var}[S \cdot H] = \sum_{i=1}^{d} \mathrm{var}[s_i h_i]$$
$$= \sum_{i=1}^{d} \left(\mathbb{E}[s_i^2 h_i^2] - \mathbb{E}[s_i h_i]^2\right)$$
$$= \sum_{i=1}^{d} \mathbb{E}[s_i^2]\,\mathbb{E}[h_i^2]$$
$$= \sum_{i=1}^{d} \mathrm{var}(s_i)\,\mathrm{var}(h_i) = d$$

Esperança e Variância do Produto Escalar

Esperança:

$$\mathbb{E}[S \cdot H] = \mathbb{E}\left[\sum_{i=1}^{d} s_i h_i\right]$$
$$= \sum_{i=1}^{d} \mathbb{E}[s_i]\,\mathbb{E}[h_i] = 0$$

Para $s_i, h_i \sim \mathcal{N}(0,1)$ i.i.d., $\mathbb{E}[s_i^2] = \mathrm{var}(s_i) = 1$ (pois $\mathbb{E}[s_i] = 0$), o que justifica os dois últimos passos.

Conclusão: $\mathrm{var}[S \cdot H]$ cresce linearmente com d.

Variância: por independência,

$$\mathrm{var}[S \cdot H] = \sum_{i=1}^{d} \mathrm{var}[s_i h_i]$$
$$= \sum_{i=1}^{d} \left(\mathbb{E}[s_i^2 h_i^2] - \mathbb{E}[s_i h_i]^2\right)$$
$$= \sum_{i=1}^{d} \mathbb{E}[s_i^2]\,\mathbb{E}[h_i^2]$$
$$= \sum_{i=1}^{d} \mathrm{var}(s_i)\,\mathrm{var}(h_i) = d$$

Scaled Dot Product Attention

Para evitar *scores* com magnitude crescente em d, normalizamos por $\sqrt{d}$:

Scaled Dot Product Attention

$$\text{score}(h_t, s_k) = \frac{h_t^\top s_k}{\sqrt{d}}, \quad k = 1, 2, \ldots, m$$

$$a_k^{(t)} = \frac{\exp(\text{score}(h_t, s_k))}{\sum_{i=1}^{m} \exp(\text{score}(h_t, s_i))}, \qquad c^{(t)} = \sum_{k=1}^{m} a_k^{(t)} s_k$$

- Escalar por $\sqrt{d}$ mantém a variância dos scores em 1
- A softmax recebe valores em uma faixa estável: gradientes mais saudáveis
- Componente fundamental do transformer (Vaswani et al., 2017)

Visão Geral

E se Aplicarmos Atenção da Sequência sobre Ela Mesma?

Até aqui usamos atenção entre o estado do decoder e os estados do encoder (RNN).

Ideia central de *self-attention*

E se utilizarmos *Dot Product Attention* nos **tokens de entrada contra eles mesmos**?

- Poderíamos codificar as relações entre tokens da entrada
- Não precisaríamos mais de RNNs como encoder

Por exemplo, considere a frase

```
"The animal didn't cross the street because it was tired"
```

A palavra `it` se refere a quais outras palavras da frase? Podemos usar *attention* para descobrir.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems*, v. 30.

Intuição: atenção como um dicionário “suave”

No tipo **dicionário**, cada **chave** (*key*) tem um **valor** (*value*), e uma **consulta** (*query*) devolve o valor da chave correspondente.

Dicionário comum

```
dic = {1: 10, 2: 50, 3: 100}
```

- `dic[2]` → `50` (chave exata)
- `dic[1.5]` → `KeyError`

Dicionário “suave”

Não dá erro: devolve uma **média dos valores ponderada pela proximidade** às chaves.

`dic[1.5]` → `30` $\left(\frac{10+50}{2}\right)$

É exatamente o *self-attention*

A **query** é comparada a todas as **keys**; quanto maior a similaridade (o *score*), mais peso o **value** correspondente recebe na média final.

Intuição: atenção como um dicionário "suave"

No tipo **dicionário**, cada **chave** (*key*) tem um **valor** (*value*), e uma **consulta** (*query*) devolve o valor da chave correspondente.

Dicionário comum

```
dic = {1: 10, 2: 50, 3: 100}
```

- `dic[2]` → `50` (chave exata)
- `dic[1.5]` → `KeyError`

Dicionário "suave"

Não dá erro: devolve uma **média dos valores ponderada pela proximidade** às chaves.

`dic[1.5]` → 30 $\left(\frac{10+50}{2}\right)$

É exatamente o *self-attention*

A **query** é comparada a todas as **keys**; quanto maior a similaridade (o *score*), mais peso o **value** correspondente recebe na média final.

Self-Attention: a Forma "Crua" Não Funciona

Aplicar dot product attention diretamente sobre os tokens da entrada $x_1, \ldots, x_m$:

$$\text{score}(x_t, x_k) = \frac{x_t\, x_k^\top}{\sqrt{d}}, \qquad a_k^{(t)} = \text{softmax}_k\big(\text{score}(x_t, x_k)\big)$$

$$c^{(t)} = \sum_{k=1}^{m} a_k^{(t)}\, x_k$$

Problema

Não há **nada para aprender** aqui: o cálculo é determinístico nas entradas. Sem parâmetros treináveis, *backpropagation* não tem o que ajustar.

Solução: introduzir três projeções lineares aprendidas para cada token, gerando **queries**, **keys** e **values**.

Queries, Keys e Values

Para cada token $x_t \in \mathbb{R}^{d_{\text{emb}}}$, projetamos em três vetores via matrizes aprendidas:

$$q_t = x_t W_q + b_q, \qquad W_q \in \mathbb{R}^{d_{\text{emb}} \times d}, \qquad q_t \in \mathbb{R}^d$$
$$k_t = x_t W_k + b_k, \qquad W_k \in \mathbb{R}^{d_{\text{emb}} \times d}, \qquad k_t \in \mathbb{R}^d$$
$$v_t = x_t W_v + b_v, \qquad W_v \in \mathbb{R}^{d_{\text{emb}} \times d_v}, \qquad v_t \in \mathbb{R}^{d_v}$$

Analogia (recuperação de informação):

- **Query**: vetor que codifica uma consulta
- **Key**: chaves de busca, comparadas às queries
- **Value**: valor retornado quando query e key são similares

A atenção passa a ser:

$$\text{score}(q_t, k_i) = \frac{q_t \, k_i^\top}{\sqrt{d}}, \qquad a_i^{(t)} = \text{softmax}_i(\text{score}(q_t, k_i))$$

$$c^{(t)} = \sum_{i=1}^{m} a_i^{(t)} \, v_i$$

Agora há parâmetros aprendidos: W_q, W_k, W_v.

Self-Attention: Exemplo Numérico (Setup)

Embeddings de entrada (dois tokens, $d_{\text{emb}} = 3$):

$$x_1 = [1,\ -3,\ 2], \qquad x_2 = [2,\ -1,\ 1]$$

Matrizes de pesos (todas com bias 0, projeção para $d = 2$):

$$W_q = \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & -2 \end{bmatrix}, \quad W_k = \begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}, \quad W_v = \begin{bmatrix} -2 & 1 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}$$

Projetamos cada token aplicando W_q, W_k, W_v; os cálculos completos vêm a seguir.

Self-Attention: Cálculo das Projeções (Queries)

Cada token é um vetor linha, multiplicado pela matriz W_q ($q_i = x_i W_q$):

$$q_1 = x_1 W_q = \begin{bmatrix} 1 & -3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 1(1)+(-3)(1)+2(0) & 1(0)+(-3)(-1)+2(-2) \end{bmatrix}$$

$$= \begin{bmatrix} -2 & -1 \end{bmatrix}$$

$$q_2 = x_2 W_q = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} 2(1)+(-1)(1)+1(0) & 2(0)+(-1)(-1)+1(-2) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & -1 \end{bmatrix}$$

Self-Attention: Cálculo das Projeções (Keys)

O mesmo procedimento com a matriz W_k ($k_i = x_i W_k$):

$$k_1 = x_1 W_k = \begin{bmatrix} 1 & -3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1(1)+(-3)(0)+2(0) & 1(1)+(-3)(-1)+2(0) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 4 \end{bmatrix}$$

$$k_2 = x_2 W_k = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 2(1)+(-1)(0)+1(0) & 2(1)+(-1)(-1)+1(0) \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 3 \end{bmatrix}$$

Self-Attention: Cálculo das Projeções (Values)

E, por fim, com a matriz W_v ($v_i = x_i W_v$):

$$v_1 = x_1 W_v = \begin{bmatrix} 1 & -3 & 2 \end{bmatrix} \begin{bmatrix} -2 & 1 \\ 0 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1(-2)+(-3)(0)+2(0) & 1(1)+(-3)(1)+2(2) \end{bmatrix}$$
$$= \begin{bmatrix} -2 & 2 \end{bmatrix}$$

$$v_2 = x_2 W_v = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} -2 & 1 \\ 0 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 2(-2)+(-1)(0)+1(0) & 2(1)+(-1)(1)+1(2) \end{bmatrix}$$
$$= \begin{bmatrix} -4 & 3 \end{bmatrix}$$

Self-Attention: Exemplo Numérico (Cálculo)

Foco no token 1: usamos $q_1 = [-2, -1]$ contra todas as keys.

Scores não normalizados (vetor linha vezes coluna):

$$q_1 k_1^\top = \begin{bmatrix} -2 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 4 \end{bmatrix} = -6, \qquad q_1 k_2^\top = \begin{bmatrix} -2 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = -7$$

Normalização por $\sqrt{d}$ (com $d = 2$):

$$-6/\sqrt{2} \approx -4.24, \qquad -7/\sqrt{2} \approx -4.95$$

Softmax sobre os scores normalizados:

$$a_1^{(1)} \approx 0.67, \qquad a_2^{(1)} \approx 0.33$$

Soma ponderada dos values:

$$c^{(1)} = 0.67 \cdot [-2, 2] + 0.33 \cdot [-4, 3] \approx [-2.66,\ 2.33]$$

O token 1 atende **principalmente** a si mesmo (0.67), mas incorpora uma fração do token 2 (0.33): a saída é uma média ponderada.

Forma Matricial: Processar Toda a Sequência de uma Vez

A implementação anterior processa cada query **de forma independente**. Como o resultado de cada query não depende das demais (no mesmo passo), podemos vetorizar tudo:

Scaled Dot Product Attention (forma matricial)

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{Q\,K^\top}{\sqrt{d}}\right)V$$

Com (X: `seq_size` × `emb_size`):

- $Q = XW_q$, W_q: `emb_size` × `d`, Q: `seq_size` × `d`
- $K = XW_k$, W_k: `emb_size` × `d`, K: `seq_size` × `d`
- $V = XW_v$, W_v: `emb_size` × `value_size`, V: `seq_size` × `value_size`

A matriz softmax($QK^\top/\sqrt{d}$) tem shape `seq_size` × `seq_size`: contém os **pesos da atenção** de cada token sobre cada token.

Forma Matricial: Projeções Q, K, V

Empilhando os dois embeddings nas linhas de X (2×3) e projetando com W_q, W_k, W_v (3×2), cada projeção fica 2×2:

$$Q = XW_q = \begin{bmatrix} 1 & -3 & 2 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & -1 \\ 0 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -1 \\ 1 & -1 \end{bmatrix}$$

$$K = XW_k = \begin{bmatrix} 1 & -3 & 2 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 3 \end{bmatrix}$$

$$V = XW_v = \begin{bmatrix} 1 & -3 & 2 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} -2 & 1 \\ 0 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} -2 & 2 \\ -4 & 3 \end{bmatrix}$$

Cada **linha** de Q, K, V é o vetor projetado de um token.

Forma Matricial: Pipeline Completo

Com as projeções do slide anterior:

$$Q = \begin{bmatrix} -2 & -1 \\ 1 & -1 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 4 \\ 2 & 3 \end{bmatrix}, \quad V = \begin{bmatrix} -2 & 2 \\ -4 & 3 \end{bmatrix}$$

Scores e normalização:

$$Q\,K^\top = \begin{bmatrix} -6 & -7 \\ -3 & -1 \end{bmatrix}, \qquad \frac{Q\,K^\top}{\sqrt{2}} \approx \begin{bmatrix} -4.24 & -4.95 \\ -2.12 & -0.71 \end{bmatrix}$$

Softmax (por linha):

$$\text{softmax} \approx \begin{bmatrix} 0.67 & 0.33 \\ 0.20 & 0.80 \end{bmatrix}$$

Saída final:

$$\text{attention}(Q, K, V) \approx \begin{bmatrix} -2.66 & 2.33 \\ -3.61 & 2.80 \end{bmatrix}$$

Refletindo sobre Self-Attention

É não linear?

Sim: $QK^\top V$ é polinomial de grau 3 nos parâmetros; a softmax já é não-linear.

Vantagem em relação a um MLP

O número de parâmetros **não escala** com o tamanho da sequência. Os mesmos W_q, W_k, W_v atendem qualquer comprimento.

Vantagem em relação a RNNs

- Muito mais fácil de **paralelizar**: tudo é matricial
- Lida melhor com **sequências longas**: dependências de longo alcance em um único passo

Bônus: explicabilidade

Os pesos da atenção formam uma matriz interpretável: dá para plotar e ver quais tokens a rede “olha”.

O Calcanhar de Aquiles: Falta Informação Posicional

Problema

Da forma como definimos, *self-attention* **não tem informação sobre a posição** dos tokens na sequência.

Em RNNs e CNNs, a informação posicional está integrada ao *viés indutivo* dos modelos, então não demandavam atenção especial. Em self-attention, não.

Consequência absurda

`"lucas comeu pastel no jantar"` = `"jantar comeu pastel no lucas"`

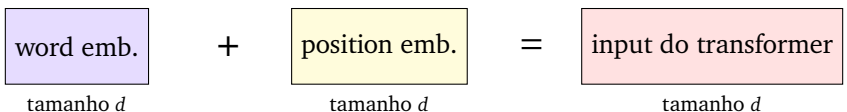
Para o modelo, ambas as frases produzem o mesmo conjunto de pesos da atenção.

Precisamos adaptar os embeddings para que carreguem informação sobre a **posição do token** na sequência.

Visão Geral

Ideia Geral do Positional Encoding

Ideia: criar uma codificação para a posição e **somá-la** ao embedding do token.



- Existem várias formas de definir/aprender embeddings posicionais
- No paper original, Vaswani et al. (2017) observaram que **embeddings posicionais aprendidos não melhoraram** resultados em relação a embeddings posicionais **fixos** (sinusoidal)

Propriedades Desejadas

Precisamos de uma matriz de posições que codifique adequadamente posições **absolutas** e **relativas** dos tokens em uma sequência.

Requisitos

- **Único por posição:** cada posição da sequência tem uma codificação distinta
- **Distâncias relativas consistentes:** o deslocamento entre duas posições é codificado de forma consistente, independentemente do tamanho da sequência
- **Generaliza para qualquer tamanho:** a codificação tem limite superior e inferior bem definidos
- **Determinística:** sem aleatoriedade na inferência

Ideia 1: Usar o Índice da Posição

Proposta: codificar a posição i pelo próprio número i (ou i/N).

Problemas

- Sem limite superior bem definido (para sequências arbitrárias)
- Os valores ficam grandes para sequências longas, dominando o embedding do token
- Normalizar por N resolve o limite, mas a mesma posição relativa passa a depender do tamanho da sequência

Não funciona.

Ideia 2: Usar Números Binários

Proposta: usar a representação binária da posição em $\lceil \log_2 N \rceil$ bits.

- Existe um **comportamento periódico** natural nos bits binários
- Bits menos significativos alternam rápido (período 2), mais significativos alternam devagar
- Isso é desejável para codificar posições

0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
⋮				
15	1	1	1	1

Problema

Redes neurais usam matemática **contínua**. Interpolar entre representações binárias não preserva a noção de distância. Também não funciona.

Ideia 3: Senos e Cossenos de Frequências Variadas

Proposta: usar uma combinação de senos e cossenos de diferentes frequências para indicar a posição.

Por que funciona

- **Limite máximo e mínimo bem definidos:** $\sin, \cos \in [-1, 1]$
- **Periodicidade multiescala:** cada dimensão tem sua própria frequência, gerando códigos únicos
- **Linearidade da rotação:** a posição (ângulo) pode ser modificada por uma transformação linear

$$\begin{pmatrix} \cos(\theta + \phi) \\ \sin(\theta + \phi) \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

Isso significa que o modelo consegue "rotacionar" as posições por uma matriz de pesos aprendida, capturando deslocamentos relativos.

Positional Encoding Sinusoidal: Fórmula

Definição (Vaswani et al., 2017)

$$P_{i,j} = \begin{cases} \sin\left(\dfrac{i}{n^{(2k/d)}}\right), & \text{se } j = 2k \\ \cos\left(\dfrac{i}{n^{(2k/d)}}\right), & \text{se } j = 2k+1 \end{cases}$$

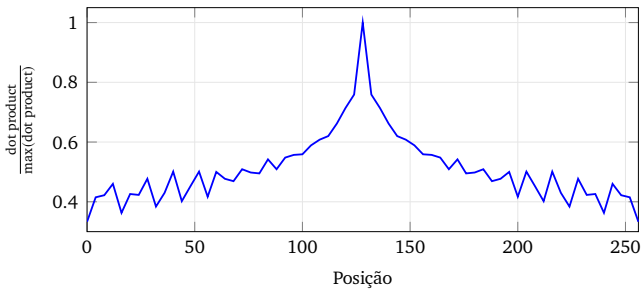
Onde:

- i: índice da posição na sequência
- j: índice da dimensão do embedding ($j = 0, \ldots, d-1$)
- d: dimensão do embedding
- n: hiperparâmetro de escala (no paper, $n = 10000$)

Cada par de dimensões $(2k, 2k+1)$ usa uma frequência diferente, gerando uma assinatura única para cada posição.

Positional Encoding: Comportamento do Produto Escalar

Uma propriedade interessante: o produto escalar entre dois positional encodings tem um pico claro na posição central e decai suavemente.



- **Pico em 1** quando comparamos uma posição com ela mesma
- **Decaimento suave** para posições próximas
- **Padrão oscilatório** característico das senoides combinadas

Isso permite ao modelo perceber proximidade relativa entre tokens.

Visão Geral

1. Atenção em RNNs
2. Scaled Dot Product Attention
3. Self-Attention
4. Positional Encoding
5. **Arquitetura Transformer**
6. Interpretabilidade

A Arquitetura Completa

Finalmente estamos prontos para abrir o transformer. Na prática, já vimos quase todos os componentes.

- À esquerda, o **encoder**: processa a sequência de entrada
- À direita, o **decoder**: gera a saída token a token

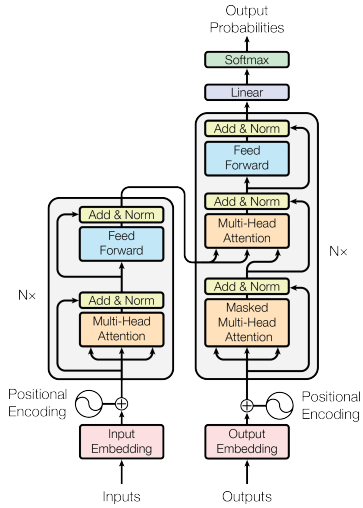


Figura de Vaswani et al., 2017. *Attention is All You Need.*

A Arquitetura Completa

Em foco: Input Embedding.

A porta de entrada do transformer: transforma índices de tokens em vetores.

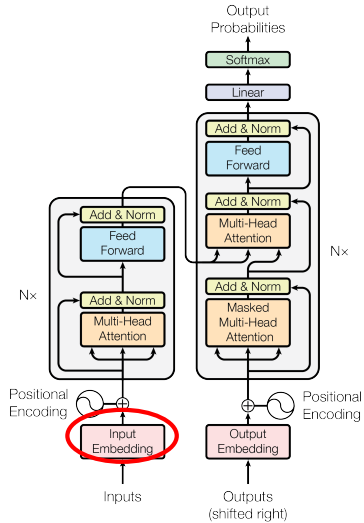


Figura de Vaswani et al., 2017. *Attention is All You Need.*

Input Embedding: uma Matriz Aprendida

Após a tokenização, cada token da sequência é um **índice inteiro** do vocabulário V.

A camada de embedding

É uma **matriz de parâmetros aprendidos**

$$E \in \mathbb{R}^{|V| \times d},$$

com uma linha por token do vocabulário: a linha t é o embedding do token de índice t. A matriz é treinada por backpropagation como qualquer outro peso da rede.

Ordem de grandeza

Com $|V| = 50\,000$ e $d = 512$, a camada de embedding tem $|V| \times d = 25{,}6$ milhões de parâmetros.

Selecionando o Embedding de um Token

Como o modelo seleciona o embedding do token de índice t? O índice vira um vetor **one-hot** $u \in \{0,1\}^{|V|}$, com 1 apenas na posição t, e o produto $u^\top E$ devolve exatamente a **linha** t de E.

Exemplo com $|V| = 4$ e $d = 3$, token de índice 3

$$u^\top E = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0.2 & -1.1 & 0.7 \\ 1.5 & 0.3 & -0.4 \\ -0.8 & 0.9 & 2.1 \\ 0.6 & -0.5 & 1.3 \end{bmatrix} = \begin{bmatrix} -0.8 & 0.9 & 2.1 \end{bmatrix} = E_{3,:}$$

Na prática, os frameworks implementam a operação como *lookup*: indexam a linha t diretamente, sem multiplicação de matrizes.

A Arquitetura Completa

Em foco: Multi-Head Attention.

O bloco de atenção que já conhecemos, agora dividido em várias “cabeças” paralelas.

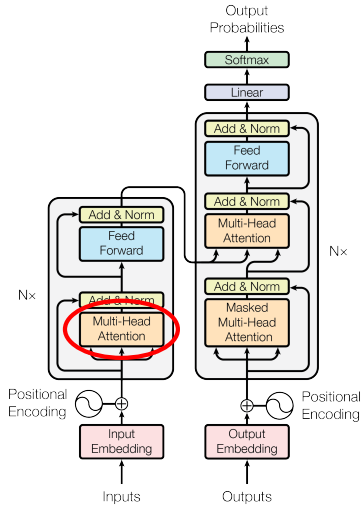
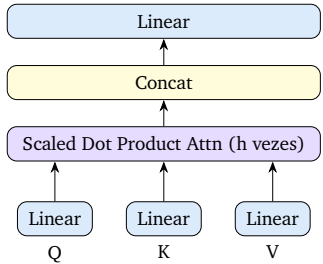


Figura de Vaswani et al., 2017. *Attention is All You Need.*

Multi-Head Attention

Ideia: separar a computação do self-attention em H "cabeças" paralelas.



Por exemplo, com $D = 512$ e $H = 8$

- Cada cabeça trabalha com dimensão $D/H = 64$
- As H cabeças computam atenção em paralelo
- As saídas são **concatenadas**
- Aplicamos uma projeção linear final

Fórmula

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W_O$$

$$\text{head}_i = \text{attention}(QW_Q^i, KW_K^i, VW_V^i)$$

A Arquitetura Completa

Em foco: Add & Norm.

Skip connections e normalização em torno de cada sub-bloco do encoder e do decoder.

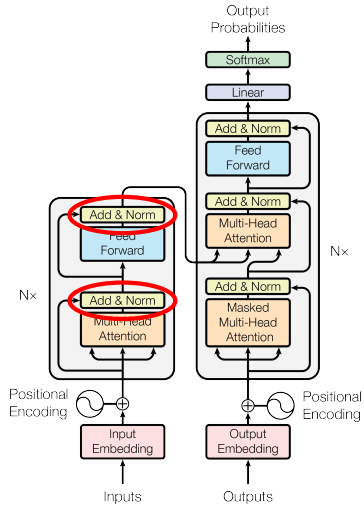


Figura de Vaswani et al., 2017. *Attention is All You Need.*

Skip Connections (Residuais)

Note no diagrama do transformer que existe uma seta passando **ao redor** do bloco *Multi-Head Attention* (e do bloco Feed Forward). Isso é uma **skip connection**.

Add & Norm

$$\text{output} = \text{LayerNorm}\big(x + \text{Subblock}(x)\big)$$

- Mantém o gradiente fluindo (mesma motivação da ResNet)
- Permite empilhar muitos blocos sem que o sinal se degrade
- "*Add*" é a soma residual; "*Norm*" é a normalização (LayerNorm)

Sem skip connections, o transformer não treinaria em profundidade.

Layer Normalization

- Similar ao **batch norm** visto anteriormente (ResNet), mas as estatísticas são computadas **por token** (não por mini-batch)
- Introduz dois parâmetros aprendidos por dimensão (escala e deslocamento)

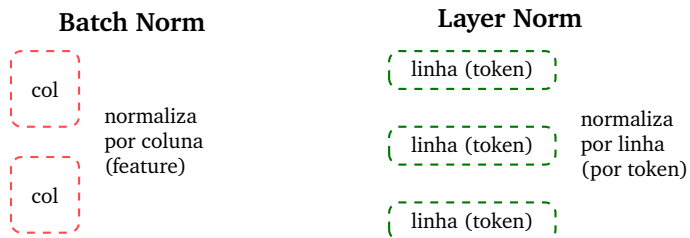
Forma fechada

Para um *token* $x_i \in \mathbb{R}^d$,

$$\mathrm{LN}(x_i) = \gamma \odot \frac{x_i - \mu_i}{\sigma_i} + \beta, \qquad \mu_i = \frac{1}{d}\sum_{k=1}^{d} x_{i,k}, \quad \sigma_i^2 = \frac{1}{d}\sum_{k=1}^{d}(x_{i,k} - \mu_i)^2.$$

Layer Norm é mais adequada para sequências de comprimento variável.

Layer Norm vs. Batch Norm



- **Batch Norm**: normaliza cada **coluna** (feature) com estatísticas do mini-batch.
- **Layer Norm**: normaliza cada **linha** (token) com as estatísticas das próprias features, independente do batch.

A Arquitetura Completa

Em foco: Feed Forward.

Um pequeno MLP aplicado a cada posição da sequência, de forma independente.

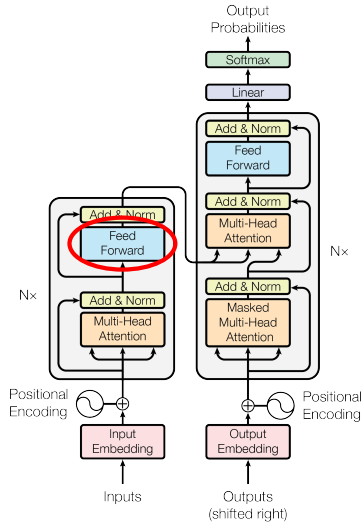


Figura de Vaswani et al., 2017. *Attention is All You Need.*

Feed-Forward Network

Após o bloco *Multi-Head Attention* e o *Add & Norm*, o transformer aplica um pequeno MLP a cada posição.

Estrutura

$$\mathrm{FFN}(x) = \mathrm{ReLU}(x\,W_1 + b_1)\,W_2 + b_2$$

- No paper original, são duas camadas com ReLU no meio
- A entrada do MLP **depende do tamanho da sequência**? Não: aplicado posição a posição (*pointwise*)
- Na atenção os tokens trocam informação entre si; no FFN cada posição processa sozinha o que coletou
- Geralmente expande a dimensão (ex.: $d \to 4d \to d$)

A Arquitetura Completa

Em foco: o bloco do encoder.

Os componentes que acabamos de ver formam um bloco, replicado N vezes.

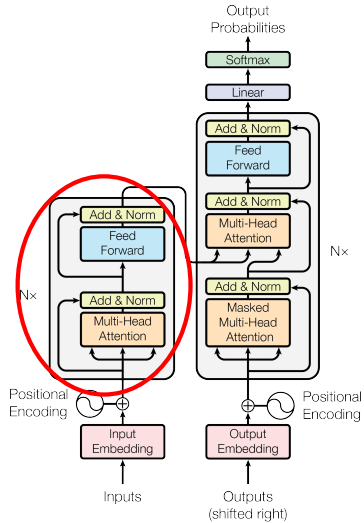


Figura de Vaswani et al., 2017. *Attention is All You Need.*

O Bloco do Encoder e a Pilha $N\times$

Cada **bloco do encoder** é a sequência:

$$\text{Multi-Head Attention} \rightarrow \text{Add \& Norm} \rightarrow \text{Feed Forward} \rightarrow \text{Add \& Norm}$$

Empilhamento

Replicamos esse bloco N vezes (no paper original, $N = 6$). Cada bloco refina a representação dos tokens, integrando informação dos demais via self-attention.

- A entrada do encoder é `Input Embedding + Positional Encoding`
- A saída do encoder é uma sequência de vetores que, conjuntamente, codificam o significado da entrada **contextualizada**
- Essa saída será consumida pelo decoder via **cross-attention**

A Arquitetura Completa

Em foco: Masked Multi-Head Attention.

O primeiro bloco de atenção do decoder só pode olhar para tokens já gerados.

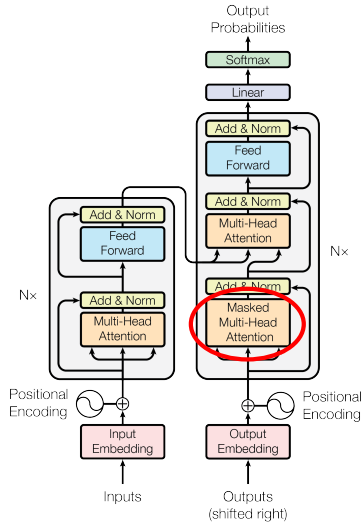


Figura de Vaswani et al., 2017. *Attention is All You Need.*

O Decoder: Masked Self-Attention

O primeiro bloco de atenção do decoder é praticamente igual ao do encoder, com uma diferença crítica: aplica uma **máscara** para olhar apenas tokens anteriores.

Por que mascarar?

Durante o treinamento (tarefa de tradução, por exemplo), o decoder vê toda a sequência de saída. Sem máscara, ele *trapaceaaria*, atendendo a tokens futuros que ainda não deveriam ter sido gerados. A máscara garante que cada posição t só atende a posições $\leq t$.

A máscara é o que torna o decoder **autoregressivo**: cada token é gerado condicionado apenas no que já foi produzido.

Mecânica da Máscara

Como aplicar a máscara:

1. Calcule os scores $QK^\top/\sqrt{d}$ (matriz `seq` $\times$ `seq`)
2. Some uma matriz de máscara M:
   - $M_{ij} = 0$ para $j \le i$ (válido)
   - $M_{ij} = -\infty$ para $j > i$ (futuro)
3. Aplique a softmax: posições com $-\infty$ recebem peso 0

0	$-\infty$	$-\infty$	$-\infty$
0	0	$-\infty$	$-\infty$
0	0	0	$-\infty$
0	0	0	0

Máscara causal triangular.

Resultado

O modelo nunca olha para tokens à direita: a atenção é estritamente **causal**.

A Arquitetura Completa

Em foco: cross-attention.

O segundo bloco de atenção do decoder consulta a saída do encoder.

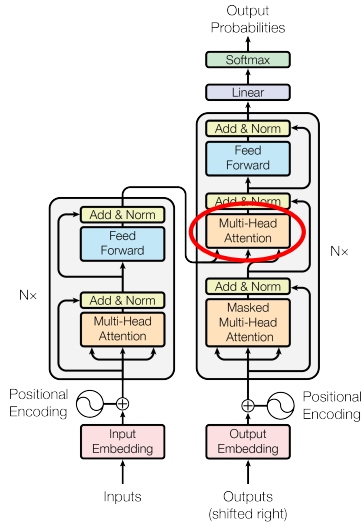


Figura de Vaswani et al., 2017. *Attention is All You Need.*

O Decoder: Cross-Attention

Após o masked self-attention, o decoder aplica um **segundo** bloco de Multi-Head Attention. Aqui há uma assimetria importante.

Cross-Attention (encoder-decoder attention)

- **Query**: vem dos estados **do decoder** (o que estou gerando agora)
- **Key** e **Value**: vêm dos estados **do encoder** (o que está na entrada)

Em outras palavras: o token alvo "olha" para a sequência de origem para decidir o que atender.

Em tradução

Ao gerar a palavra "cat", a query do decoder consulta as keys do encoder e descobre que deve dar mais peso ao value associado ao token original que representa "gato" na sequência de origem.

A Arquitetura Completa

Em foco: a cabeça de saída.

Uma projeção linear seguida de softmax produz a distribuição sobre o próximo token.

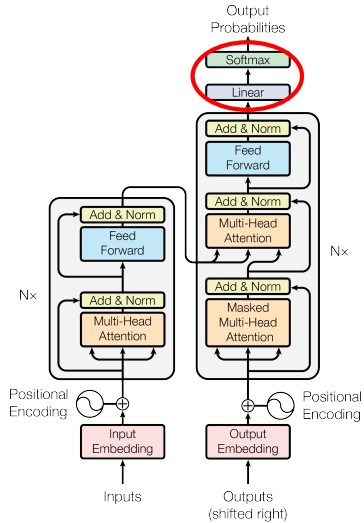


Figura de Vaswani et al., 2017. *Attention is All You Need.*

O Decoder: Feed-Forward e Saída

Bloco completo do decoder

1. Masked Multi-Head Attention → Add & Norm
2. Cross-Attention (Q do decoder, K/V do encoder) → Add & Norm
3. Feed Forward → Add & Norm

Replicado N vezes (no paper, $N = 6$).

Cabeça final

Após a pilha de blocos do decoder:

- Uma **Linear** projeta cada vetor de saída para a dimensão do vocabulário $|V|$
- Uma **Softmax** produz uma distribuição de probabilidade sobre o próximo token

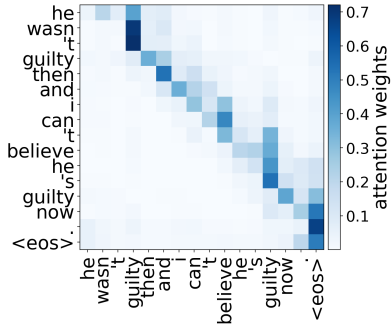
A geração ocorre token a token: amostramos (ou pegamos o argmax) do próximo token, adicionamos à entrada do decoder, e repetimos.

Visão Geral

1. Atenção em RNNs
2. Scaled Dot Product Attention
3. Self-Attention
4. Positional Encoding
5. Arquitetura Transformer
6. **Interpretabilidade**

A Matriz de Atenção como Janela para o Modelo

Os mecanismos de atenção são **especialmente fáceis de inspecionar**: podemos plotar os pesos da atenção e ver o que o modelo está "olhando".



Exemplo real: self-attention no encoder de um modelo de tradução. A diagonal predominante mostra que cada token atende sobretudo a si mesmo e aos vizinhos.

Figura de Voita et al., 2019. *Analyzing multi-head self-attention.*

Cabeças Especializadas

Pesquisas (Voita et al., 2019; entre outros) mostram que algumas **attention heads** se especializam em fenômenos linguísticos conhecidos.

Exemplos de especialização observada

- **Posicional:** a cabeça atende sempre ao token na mesma posição relativa
- **Concordância verbo-sujeito:** liga formas conjugadas ao seu sujeito
- **Objeto-verbo:** liga objetos diretos ao verbo correspondente
- **Palavras raras:** cabeças que "buscam" tokens raros para resolver ambiguidade

Conclusão

Não basta empilhar atenção: as cabeças aprendem **papéis distintos**. Algumas podem ser podadas com perda mínima, sugerindo redundância controlável.

Leituras Recomendadas

- Kazemnejad, A. *Transformer Architecture: The Positional Encoding*. kazemnejad.com/blog/transformer_architecture_positional_encoding/
- Alammar, J. *The Illustrated Transformer*. jalammar.github.io/illustrated-transformer/

Referências

Artigos principais:

- Vaswani, A. et al. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems*, v. 30.
- Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473*.
- Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective Approaches to Attention-based Neural Machine Translation. *arXiv:1508.04025*.

Análise e interpretabilidade:

- Voita, E., Talbot, D., Moiseev, F., Sennrich, R., & Titov, I. (2019). Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. *arXiv:1905.09418*.