

UNIVERSIDADE DE SÃO PAULO  
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA  
BACHARELADO EM MATEMÁTICA APLICADA COMPUTACIONAL

**Anatomia do ChatGPT**

*Aspectos matemáticos e computacionais  
dos grandes modelos de linguagem*

Milton Leal

TRABALHO DE FORMATURA

MAP 2040

Orientador: Prof. Renato Vicente

São Paulo  
2024

*O conteúdo deste trabalho é publicado sob a licença CC BY 4.0  
(Creative Commons Attribution 4.0 International License)*

*Palavras são números disfarçados.*

# Agradecimentos

Desde criança, meus pais me ensinaram a caminhar pela estrada do saber. Nada de tudo de bom que acontecera em minha vida teria ocorrido não fosse pela curiosidade em estudar as coisas do mundo. Inicialmente foram as palavras, é verdade. Mais tarde vieram os números. Com eles, surgiram os amigos do Instituto de Matemática e Estatística da Universidade de São Paulo, a saber: Fadel, Lina, Rafael, Richard, Guilherme, Lucka, Kauê, Peterson e tantos mais que interagi e troquei conhecimento.

Aos professores e professoras, um enorme obrigado pela paciência em ensinar a beleza da linguagem matemática. Em especial ao professor Renato Vicente, obrigado pelas sessões entrópicas e discussões sobre o futuro da humanidade.

À minha esposa, Amanda, reconheço a dívida imensurável pelos vários, longos ePersistentes momentos de ausência. Muito obrigado por esperar.

Dedico esta empreitada à memória de meu pai.

## Resumo

Milton Leal. **Anatomia do ChatGPT: Aspectos matemáticos e computacionais dos grandes modelos de linguagem.** Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

O progresso dos Grandes Modelos de Linguagem, particularmente os baseados na arquitetura de redes neurais do tipo *Transformer*, impulsionou avanços significativos no campo da Inteligência Artificial e na subárea do Processamento de Linguagem Natural. Este trabalho dedica-se à exploração dos componentes matemáticos e computacionais que formam os modelos baseados no *decoder* do *Transformer*, conhecidos como *Generative Pre-trained Transformer* (GPT), que sustentam poderosas ferramentas de geração e compreensão de textos, como o ChatGPT. A análise é complementada por uma implementação em Python de um GPT em pequena escala treinado na obra completa do escritor brasileiro Machado de Assis, ilustrando o potencial deste tipo de modelo para a generalização da linguagem natural.

**Palavras-chave:** Redes Neurais. Transformador Pré-treinado Generativo. Grandes Modelos de Linguagem.

# Abstract

Milton Leal. **Anatomy of ChatGPT: Mathematical and Computational Aspects of Large Language Models.** Undergraduate Thesis (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

The progress of Large Language Models, particularly those based on the neural network architecture known as the Transformer, has propelled significant advances in the field of Artificial Intelligence and in the subarea of Natural Language Processing. This work is dedicated to exploring the mathematical and computational components that constitute models based on the Transformer's decoder, known as Generative Pre-trained Transformer (GPT), which support powerful tools for text generation and understanding, such as ChatGPT. The analysis is complemented by an implementation in Python of a small-scale GPT trained on the complete works of the Brazilian writer Machado de Assis, illustrating the potential of this type of model for the generalization of natural language.

**Keywords:** Neural Networks. Generative Pre-trained Transformers. Large Language Models.

# Listas de abreviaturas

IA	( <i>Inteligência Artificial</i> )
LLM	( <i>Large Language Model</i> )
GPT	( <i>Generative Pre-trained Transformer</i> )
BERT	( <i>Bidirectional Encoder Representations from Transformers</i> )
RHLF	( <i>Reinforcement Learning From Human Feedback</i> )
BPE	( <i>Byte Pair Encoding</i> )
FFNN	( <i>Feed-Forward Neural Network</i> )
RNN	( <i>Recurrent Neural Network</i> )
LSTM	( <i>Long Short-Term Memory</i> )
CNN	( <i>Convolutional Neural Network</i> )
GPU	( <i>Graphics Processing Unit</i> )
API	( <i>Application Programming Interface</i> )
RBRM	( <i>Rule-Based Reward Model</i> )
BOS	( <i>Beginning Of Sentence</i> )
EOS	( <i>End Of Sentence</i> )
PAD	( <i>Padding</i> )
ReLU	( <i>Rectified Linear Unit</i> )
SGD	( <i>Stochastic Gradient Descent</i> )
Adam	( <i>Adaptive Moment Estimation</i> )

# Listas de símbolos

$\mathcal{D}$	Conjunto de dados (sequências de textos)
$\mathbf{x}$	Uma sequência de texto composta por <i>tokens</i>
$x_i$	um <i>token</i> da sequência
$P(\mathbf{x})$	Probabilidade da sequência $\mathbf{x}$
$\mathcal{V}$	Vocabulário do corpus de treinamento
$N_{\mathcal{V}}$	Tamanho do vocabulário
$B$	Tamanho do lote de sequências ( <i>batch</i> )
$T$	Comprimento máximo das sequências ou janela de contexto
$t$	Posição do <i>token</i> na sequência
$d$	Tamanho do vetor de <i>embedding</i>
$W_{emb}$	Matriz de <i>embeddings</i>
$i$	Posição do vetor <i>embeddings</i>
$W^Q$	Matriz de pesos das consultas ( <i>queries</i> ) da cabeça de Atenção
$W^K$	Matriz de pesos das chaves ( <i>keys</i> ) da cabeça de Atenção
$W^V$	Matriz de pesos dos valores ( <i>values</i> ) da cabeça de Atenção
$Q$	Representa a sequência de entrada transformada pela matriz de pesos das consultas
$K$	Representa a sequência de entrada transformada pela matriz de pesos das chaves
$V$	Representa a sequência de entrada transformada pela matriz de pesos dos valores
$d_k$	Dimensão da matriz de chaves
$h$	Número de cabeças de Atenção quando considerada a Atenção Paralela
$W_i^Q$	Matriz de consultas da Atenção Paralela
$W_i^K$	Matriz de chaves da Atenção Paralela
$W_i^V$	Matriz de valores da Atenção Paralela
$W^O$	Matriz que projeta as múltiplas cabeças de Atenção para a dimensão do modelo
$N_x$	Número de blocos de atenção
$b$	Termo de viés ( <i>bias</i> )
$W_1$	Primeira camada linear da FFNN
$W_2$	Segunda camada linear da FFNN

$F_1(\mathbf{x})$	Função da primeira camada oculta da FFNN
$F_2(\mathbf{x})$	Função da segunda camada oculta da FFNN
$d_{ff}$	Dimensão da camada oculta da FFNN
$\mu$	Média de um conjunto de dados
$\sigma^2$	Variância de um conjunto de dados
$\sigma$	Desvio padrão de um conjunto de dados
$\epsilon$	Número muito pequeno
$\gamma$	Parâmetro da camada de normalização
$\beta$	Parâmetro da camada de normalização
$G$	Matriz cujas entradas são variáveis de Bernoulli
$\odot$	Multiplicação elemento a elemento entre matrizes
$z$	Matriz de logitos
$W^f$	Matriz de pesos da camada linear final
$\Theta$	Conjunto de parâmetros do modelo
$\mathcal{L}$	Função de perda de entropia cruzada
$\hat{P}(\mathbf{x})$	Distribuição a ser estimada pelo modelo
$\eta$	Taxa de aprendizagem do otimizador
$\nabla_\Theta$	Gradiente com relação aos parâmetros
$\hat{m}$	Estimativa corrigida do primeiro momento dos gradientes
$\hat{v}$	Estimativa corrigida do segundo momento dos gradientes
$\frac{\partial \mathcal{L}}{\partial a}$	Gradiente da função de perda com relação à ativação da camada $a$
$\frac{\partial a}{\partial \theta}$	Gradiente da ativação $a$ com relação aos parâmetros

# Listas de figuras

1.1	Arquitetura <i>Transformer</i> proposta por VASWANI <i>et al.</i> , 2017 . . . . .	2
1.2	Arquitetura de um GPT protótipo com apenas um bloco de atenção. Elaboração do autor. . . . .	3
2.1	Frequência relativa em relação à ordem das palavras. Retirado de SHANNON, 1951. . . . .	6
2.2	Representação artesanal das sentenças de uma língua. Retirado de CHOMSKY, 1957. . . . .	7
2.3	Interação com Eliza. Autor desconhecido ( <i>Wikimedia Commons</i> ). . . . .	7
2.4	O processador acústico é projetado para atuar como um foneticista, transcrevendo a forma de onda da fala em uma sequência de símbolos fonéticos. Enquanto isso, o decodificador linguístico traduz a sequência fonética possivelmente distorcida em uma sequência de palavras. Retirado de L. R. BAHL <i>et al.</i> , 1983. . . . .	8
2.5	Exemplo de uma rede com 8 unidades de entrada, 4 unidades de saída e 2 blocos de células de memória com tamanho 2. Retirado de HOCHREITER e SCHMIDHUBER, 1997. . . . .	8
2.6	Modelo de Linguagem Neural: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ , onde $g$ é a rede neural e $C(i)$ é o <i>embedding</i> da $i$ -ésima palavra. Retirado de BENGIO <i>et al.</i> , 2003. . . . .	9
2.7	Exemplo que ilustra o alinhamento de palavras obtido pelo <i>RNNsearch-50</i> entre a sentença fonte em inglês e a tradução gerada em francês. Cada pixel no gráfico indica o peso $\alpha_{ij}$ , que representa o grau de atenção da $i$ -ésima palavra na sentença alvo (francês) para a $j$ -ésima palavra na sentença fonte (inglês). A escala de cores varia do preto (0) ao branco (1), indicando o peso da anotação correspondente. Retirado de BAHDANAU <i>et al.</i> , 2014. . . . .	10

2.8	À esquerda, arquitetura <i>decoder</i> do <i>Transformer</i> e objetivos de treinamento utilizados. À direita, transformações de entrada para etapa de <i>fine-tuning</i> em diferentes tarefas. Note que o GPT-1 possui 12 blocos de atenção. Retirado de RADFORD, NARASIMHAN <i>et al.</i> , 2018. . . . .	11
2.9	Arquiteturas e hiperparâmetros dos modelos treinados e descritos no artigo do GPT-3. Todos os modelos foram treinados com um total de 300 bilhões de tokens. Retirado de BROWN <i>et al.</i> , 2020. . . . .	12
2.10	Exemplos ilustrativos de <i>prompts</i> retirados do artigo do ChatGPT inspirados em casos reais de uso. Inclui casos de uso como <i>brainstorming</i> e geração de texto, com exemplos de <i>prompts</i> como "Listar cinco ideias para recuperar o entusiasmo pela minha carreira" e "Escrever um conto em que um urso vai à praia, faz amizade com uma foca e depois retorna para casa", além de sumários para peças da <i>Broadway</i> e esboços de comerciais para essas peças. Retirado de OUYANG <i>et al.</i> , 2022. . . . .	12
2.11	Desempenho do GPT em exames acadêmicos e profissionais. A simulação segue as condições e critérios de avaliação dos exames reais, ordenados com base no desempenho do GPT-3.5. O GPT-4 supera o GPT-3.5 na maioria dos exames testados. Retirado de OPENAI, 2023. . . . .	13
2.12	Exemplo de <i>prompt</i> e respostas para melhorias na recusa de categorias proibidas. O exemplo mostra a evolução da resposta do modelo GPT-4, desde uma explicação inicial que poderia ser perigosa até uma recusa atualizada que segue diretrizes de segurança e legalidade. Retirado de OPENAI, 2023. . . . .	14
5.1	À esquerda, arquitetura CBOW, que prevê a palavra do centro dado um contexto adjacente de $n$ tokens. À direita, a arquitetura Skip-Gram, que prevê o entorno de tamanho $n$ tokens de uma palavra que esteja no centro.	23
6.1	No eixo da ordenada, temos a posição de cada token na sequência de entrada, aqui hipoteticamente com janela de contexto de tamanho 100. No eixo das abcissas, temos a dimensão ou índice de cada posição do vetor de <i>embeddings</i> , arbitrariamente escolhido como 512. Para cada token da sequência, adiciona-se ao vetor de <i>embedding</i> correspondente a linha da matriz de codificação posicional que corresponde à posição desse token na sequência. . . . .	29
7.1	O diagrama, retirado de VASWANI <i>et al.</i> , 2017, mostra o fluxo das operações realizadas no cálculo da Atenção por Produto Escalar Dimensionado ( <i>Scaled Dot-Product Attention</i> ) com mascaramento. . . . .	33

7.2	O diagrama de VASWANI <i>et al.</i> , 2017 resume as operações da Atenção Paralela. . . . .	34
7.3	À esquerda, exemplo retirado de VASWANI <i>et al.</i> , 2017 mostra duas cabeças de Atenção (nota: cada linha colorida representa uma cabeça de Atenção) atendendo de maneira diferente à palavra "its". À direita, temos apenas uma cabeça de Atenção com a opacidade da cor representando maiores ou menores pesos de Atenção. . . . .	35
8.1	No diagrama, retirado de MUKUL RATHI, 2018, temos a arquitetura de uma FFNN simples. No GPT, essa estrutura é muito maior, potencialmente com milhares de nós intermediários. . . . .	38
9.1	No diagrama, retirado de HAO LI, 2018, à esquerda temos uma superfície da função de perda de um modelo de imagem sem a presença de conexões residuais. À direita, a superfície da função de perda da mesma rede com a adição das conexões, muito mais suave e fácil de ser otimizada. . . . .	43
12.1	No diagrama, retirado de OUYANG <i>et al.</i> , 2022 vemos a comparação da eficácia de diferentes abordagens de alinhamento em modelos GPT-3 de vários tamanhos. A primeira linha debaixo para cima mostra o GPT-3 pré-treinado, em seguida o mesmo modelo com a técnica de prompting (in-context learning). A terceira linha mostra a performance do GPT-3 após o refinamento por instrução. Por último, as duas linhas de cima do gráfico mostram a combinação desta última técnica com o RLHF em diferentes cenários envolvendo o algoritmo de otimização. . . . .	54
A.1	Machado de Assis (1839-1908) é considerado um dos maiores escritores em língua portuguesa. (Foto: Arquivo Nacional. Autor desconhecido) . . . . .	57
A.2	Resultados da função de perda nos conjuntos de treino e validação. . . . .	58
A.3	Nosso modelo conseguiu aproveitar quase toda a memória disponível da GPU que possuímos para o pré-treinamento. . . . .	58

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Perspectiva histórica</b>	<b>6</b>
<b>3</b>	<b>Modelagem Probabilística de Linguagem</b>	<b>15</b>
<b>4</b>	<b>"Tokenização" e Indexação</b>	<b>17</b>
4.1	<i>Byte Pair Encoding</i>	17
4.2	Indexação	19
4.3	Janela de Contexto	19
<b>5</b>	<b>Embeddings</b>	<b>22</b>
5.1	Calculando <i>embeddings</i>	23
5.2	Aplicação no GPT	26
<b>6</b>	<b>Codificação Posicional</b>	<b>28</b>
6.1	Calculando a Codificação Posicional	29
6.2	Aplicação no GPT	30
<b>7</b>	<b>Mecanismo de Atenção</b>	<b>31</b>
7.1	Intuição	31
7.2	Cálculo da Atenção	32
7.3	Atenção Paralela	34
7.4	Aplicação no GPT	36
<b>8</b>	<b>Rede Neural <i>Feed-Forward</i></b>	<b>38</b>
8.1	Estrutura da FFNN	39
8.2	Aplicação no GPT	39
<b>9</b>	<b>Componentes Auxiliares</b>	<b>41</b>

9.1	Camada de Normalização . . . . .	41
9.2	Conexões Residuais . . . . .	42
9.3	<i>Dropout</i> . . . . .	43
<b>10</b>	<b>Gerando Probabilidades</b>	<b>45</b>
10.1	Camada Linear . . . . .	45
10.2	<i>Softmax</i> . . . . .	46
<b>11</b>	<b>Pré-treinamento</b>	<b>47</b>
11.1	Função de Perda . . . . .	48
11.2	Otimização dos Parâmetros . . . . .	49
<b>12</b>	<b>Alinhamento</b>	<b>53</b>
12.1	Refinamento por Instrução . . . . .	53
12.2	Modelo de Recompensa . . . . .	54
12.3	Aprendizado por Reforço com Feedback Humano . . . . .	54
<b>13</b>	<b>Conclusão</b>	<b>55</b>
<b>Apêndices</b>		
<b>A</b>	<b>Implementação em Python</b>	<b>57</b>
<b>Referências</b>		<b>67</b>

# Capítulo 1

## Introdução

O surgimento do ChatGPT - um *chatbot* capaz de manter conversações, gerar textos coerentes e, em certa medida, raciocinar - marcou um significativo avanço na subárea da Inteligência Artificial (IA) conhecida como Processamento de Linguagem Natural (NLP, do inglês *Natural Language Processing*). Segundo **BUBECK et al., 2023**, inaugura-se uma nova era de modelos cuja inteligência geral é superior à anterior.

Essencialmente, o ChatGPT é um modelo probabilístico, ou seja, um robô capaz de atribuir probabilidades para a próxima palavra de uma frase com base no contexto anterior. Ele adquire essa capacidade após ser treinado com um conjunto de dados de texto diversificado e extenso. Assim, ele aprende que a sentença "chocolate é \_\_" tem maior probabilidade de ser completada com a palavra "doce" do que com a palavra "azedo".

Divulgada em novembro de 2022 pela empresa americana OpenAI, a aplicação havia alcançado a marca de 100 milhões de usuários em apenas dois meses após o lançamento. Tamanha popularização impulsionou várias empresas a começarem a trabalhar na construção de interfaces entre humanos e máquinas que utilizam semelhantes modelos de IA para resolver problemas em diferentes áreas, como atendimento ao cliente, programação, educação, saúde e muitas outras.

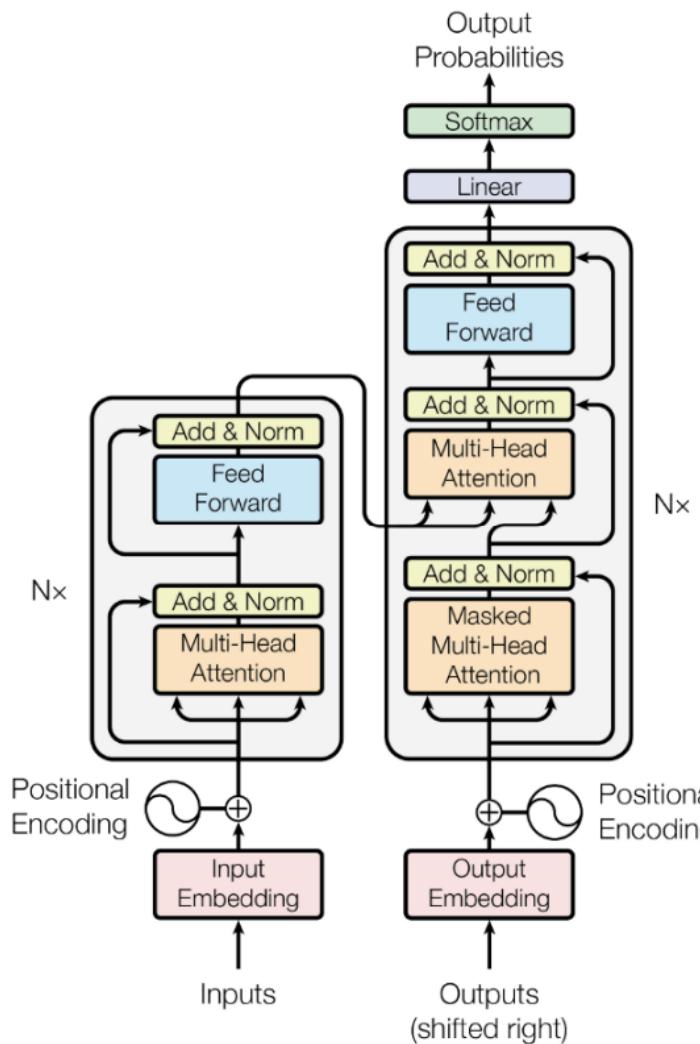
Devido aos bilhões de parâmetros que possui e também à quantidade de texto utilizado em seu treinamento, o ChatGPT é denominado um Grande Modelo de Linguagem (LLM, do inglês *Large Language Model*). Do seu nome ainda decorre a classe de modelos de aprendizado profundo (*deep learning*) que foi utilizada, chamada Transformador Pré-treinado Generativo (GPT, do inglês *Generative Pre-trained Transformer*) **LIU et al., 2018 RADFORD, NARASIMHAN et al., 2018 RADFORD, WU et al., 2019 BROWN et al., 2020 OUYANG et al., 2022 OPENAI, 2023**.

Os modelos do tipo GPT baseiam-se na arquitetura de redes neurais conhecida como *Transformer*, introduzida pela primeira vez em 2017 por cientistas do Google por meio do artigo seminal “Atenção é tudo o que você precisa” (*Attention is all you need*) **VASWANI et al., 2017 JAKOB USZKOREIT, 2017**.

Dentre as principais inovações das redes do tipo *Transformer*, está a incorporação do Mecanismo de Atenção (*Attention Mechanism*), introduzido alguns anos antes. Com isso, os modelos ganharam a capacidade de processar simultaneamente cada palavra de uma

sequência de texto em relação a todas as outras da mesma sequência. Computacionalmente, isso permitiu uma paralelização mais eficiente dos dados de entrada em comparação com modelos precedentes, como as redes neurais recorrentes KAMATH *et al.*, 2022.

É importante salientar que a arquitetura do *Transformer* foi concebida originalmente para tarefas de tradução automática de linguagem. A versão dita "original" é composta por dois blocos principais denominados: codificador (*encoder*) e decodificador (*decoder*). Na Figura 1.1, o conjunto à esquerda composto por  $N_x$  blocos de atenção representa o *encoder*. À direita, temos o *decoder*, composto por outro conjunto de blocos de mesmo tamanho com algumas modificações que exploraremos ao longo deste trabalho.



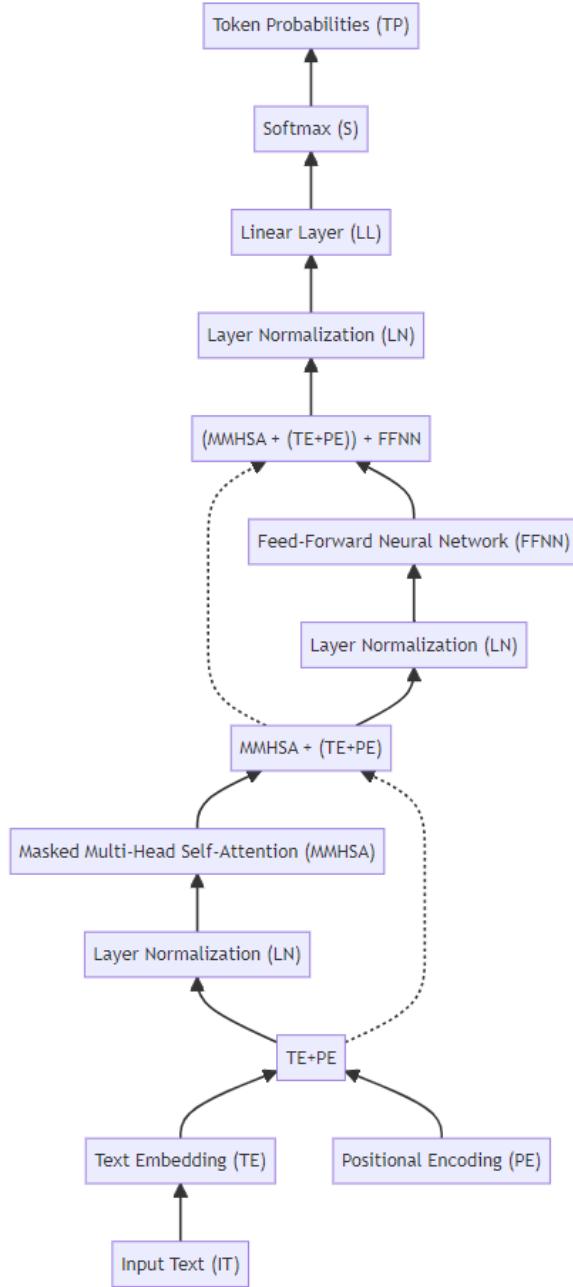
**Figura 1.1:** Arquitetura Transformer proposta por VASWANI *et al.*, 2017

*Transformers* são categorizados predominantemente em três subclasses: modelos *encoder-decoder*, como o *Transformer* original, modelos que utilizam apenas o *encoder* e modelos baseados exclusivamente no *decoder* ROTHMAN, 2021.

Os LLMs que empregam somente a componente *encoder* são associados à arquitetura BERT (do inglês *Bidirectional Encoder Representations from Transformers*), que também foi introduzida pelo Google DEVLIN *et al.*, 2018. Em contrapartida, os modelos do tipo GPT,

que são otimizados para a tarefa de geração de textos e compreensão da linguagem natural, são construídos utilizando apenas o conjunto *decoder*.

O presente trabalho foca nos aspectos matemáticos e computacionais de cada uma das peças que constituem o *decoder*, conforme ilustra o diagrama da Figura 1.2.



**Figura 1.2:** Arquitetura de um GPT protótipo com apenas um bloco de atenção. Elaboração do autor.

É importante mencionar que, como os elementos do *encoder* são semelhantes aos encontrados no *decoder*, o estudo deste pode oferecer uma visão abrangente da arquitetura original do *Transformer*.

Para esmiuçar o ChatGPT, utilizaremos como exemplo uma versão simplificada do

GPT, que se assemelha às primeiras gerações do modelos (GPT-1, GPT-2) criadas pela OpenAI. Tal versão foi codificada com a biblioteca *PyTorch* da linguagem *Python* e pode ser encontrada no Apêndice A deste trabalho. Ela é baseada na implementação proposta por Andrej Karpathy, ex-chefe de inteligência artificial da Tesla e membro fundador da OpenAI ANDREJ KARPATY, 2023.

Ao longo deste texto, utilizaremos os termos GPT e ChatGPT como sinônimos. No entanto, é importante que o leitor saiba que o ChatGPT é um produto da OpenAI, que é muito maior e mais complexo, com detalhes não divulgados pela empresa. Além disso, é válido mencionar que, até janeiro de 2024 (data de conclusão deste trabalho), o usuário assinante do ChatGPT podia selecionar dentre dois modelos (GPT-3.5 e GPT-4) para interagir, sendo o primeiro mais rápido e menos capaz e o segundo mais lento e mais poderoso.

Quando do lançamento das primeiras versões dos modelos, a companhia tinha como missão desenvolver sob o espírito da filosofia de código aberto (*open source*) — daí o nome OpenAI — e, portanto, os detalhes daquelas implementações são amplamente conhecidos e servem como base para o desenvolvimento deste texto.

Contudo, a partir de 2019, a OpenAI formou uma entidade com fins lucrativos e passou a desenvolver seus principais modelos a portas fechadas. Por isso, pouco se sabe sobre o seu modelo mais capaz, o GPT-4.

A construção de aplicações como o ChatGPT envolve três principais etapas: pré-treinamento (*pre-training*), refinamento (*fine-tuning*) e aprendizado por reforço com *feedback* humano (RHLF, do inglês *Reinforcement Learning From Human Feedback*). Estas duas últimas fases podem ser englobadas como o processo de "alinhamento" (*alignment*) do modelo. Essa combinação de técnicas possibilitou que o ChatGPT apresentasse desempenho superior em diversas tarefas de compreensão e geração de linguagem natural, simulando raciocínio. SEBASTIAN RASCHKA, 2023.

O foco principal deste trabalho reside na fase de pré-treinamento. Não obstante, dedicamos um breve capítulo para explicar de maneira geral a importância das outras duas etapas. Além disso, no Apêndice A, mostramos resultados do pré-treinamento do protótipo do GPT utilizando como conjunto de dados a obra completa do escritor Machado de Assis.

Este trabalho busca contribuir para a literatura acadêmica ao fornecer uma descrição abrangente e detalhada do GPT, um dos principais modelos utilizados em NLP e considerado estado da arte em várias tarefas de NLP. Por meio de exploração teórica e prática, o objetivo é fornecer uma compreensão clara da arquitetura *decoder* do *Transformer* e seu potencial no campo da IA.

Este trabalho está estruturado da seguinte forma:

**Perspectiva Histórica:** oferece um panorama da evolução do campo de NLP, com ênfase na transição para abordagens de aprendizado de máquina profundo.

**Modelagem de Linguagem:** aborda a modelagem probabilística do GPT.

**Tokenização e Indexação:** detalha o processo de transformar o texto em *tokens*, com destaque ao método de codificação por pares de *bytes* (BPE, do inglês *Byte Pair Encoding*),

além de explicar a necessidade de indexá-los.

**Embeddings:** discute a importância crucial da técnica de representar os *tokens* como vetores densos em espaços vetoriais de alta dimensão.

**Positional Encoding:** explica a necessidade e metodologia para adicionar informações posicionais aos vetores de *embedding*.

**Mecanismo de Atenção:** descreve em detalhes a importância e o funcionamento deste algoritmo que é considerado por muitos como o "coração" do *Transformer*.

**Feed-Forward Neural Network:** examina a estrutura e função das redes neurais dentro do GPT, elucidando como elas contribuem na transformação da informação.

**Componentes Auxiliares:** aborda o papel e os cálculos associados à normalização de camadas, enfatizando sua importância para a estabilidade e eficiência do modelo; explica o significado e a operacionalização das conexões residuais no treinamento; e expõe a utilização da técnica de *dropout* para evitar o sobreajuste (*overfitting*) do modelo.

**Gerando Probabilidades:** discute a camada linear final do modelo e a utilização da função *Softmax* para a obtenção da distribuição de probabilidades dos *tokens*.

**Pré-treinamento:** descreve as etapas e técnicas utilizadas para o treinamento inicial do GPT, incluindo a função de perda a ser otimizada e técnicas de otimização.

**Alinhamento:** discorre sobre as cruciais etapas que permitem transformar um modelo pré-treinado em uma aplicação com utilidade para os seres humanos com o emprego das etapas de *fine-tuning* e RHLF.

**Conclusão:** sumariza as principais discussões realizadas neste trabalho e sugere possíveis direções para futuras pesquisas.

**Apêndice:** contém a implementação da versão simplificada do GPT e os resultados do treinamento realizado com base na obra de Machado de Assis.

# Capítulo 2

## Perspectiva histórica

Em 1950, o "pai da teoria da informação", o matemático Claude Shannon, investigava e propunha experimentos que envolviam a previsão da próxima letra de um texto tendo o contexto prévio dado como conhecido. Em [SHANNON, 1951](#), baseado nas medidas estatísticas de Entropia e Redundância, ele estabelece a quantidade média de informação produzida por cada letra de um texto em uma língua.

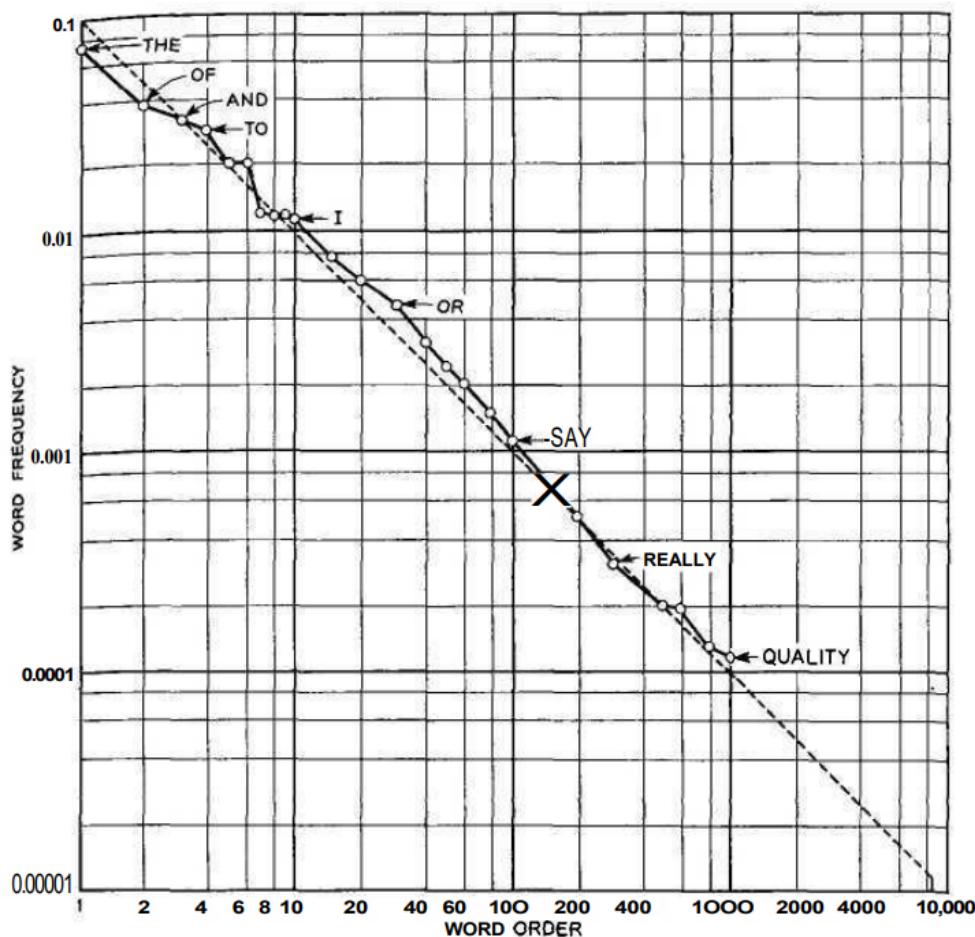
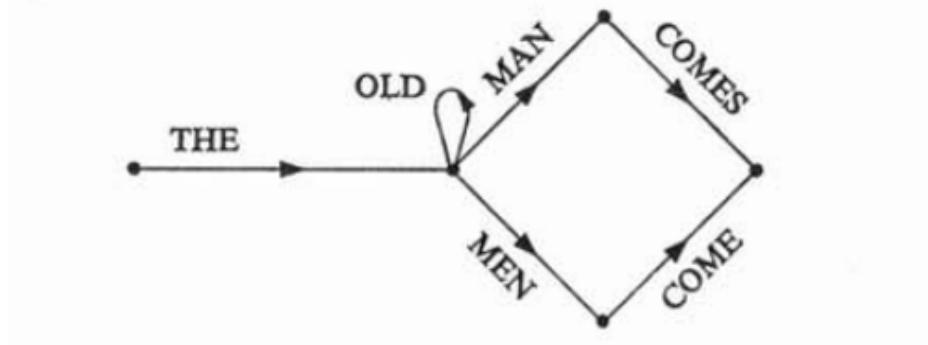


Figura 2.1: Frequência relativa em relação à ordem das palavras. Retirado de [SHANNON, 1951](#).

Ainda na década de 50, outras vertentes baseavam-se fundamentalmente em abordagens simbólicas. Neste paradigma, linguistas e cientistas da computação criavam manualmente conjuntos de regras gramaticais e lexicais para modelar a linguagem.



**Figura 2.2:** Representação artesanal das sentenças de uma língua. Retirado de *CHOMSKY, 1957*.

Outro exemplo marcante desta fase é o chatbot ELIZA, desenvolvido entre 1964 e 1966 por [WEIZENBAUM, 1966](#). ELIZA agia como uma terapeuta, empregando técnicas simples de correspondência de padrões para identificar palavras-chave na entrada do usuário e formular respostas apropriadas. Se ele dissesse "eu me sinto triste", ELIZA identificava a palavra "triste" e gerava uma resposta como, por exemplo, "por que você se sente triste?".

```
Welcome to
EEEEEELL      IIII    ZZZZZZ   AAAAAA
EE     LL      II      ZZ   AA   AA
EEEEEELL      II      ZZZ   AAAAAAA
EE     LL      II      ZZ   AA   AA
EEEEEELLLLL IIII    ZZZZZZ   AA   AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

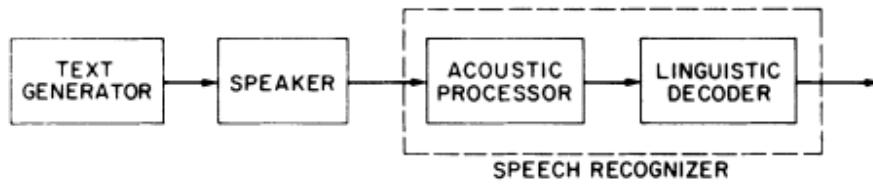
ELIZA: Is something troubling you ?
YOU: Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU: They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU: Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU: He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU: It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

**Figura 2.3:** Interação com Eliza. Autor desconhecido (Wikimedia Commons).

ELIZA foi um dos primeiros programas a serem submetidos ao Teste de Turing, inicialmente proposto por [TURING, 1950](#). Neste teste, um humano e uma máquina se comunicam em linguagem natural por meio de um intermediário. Se o intermediário não conseguir distinguir qual é a máquina, considera-se que a máquina passou no teste, evidenciando características de inteligência artificial avançada.

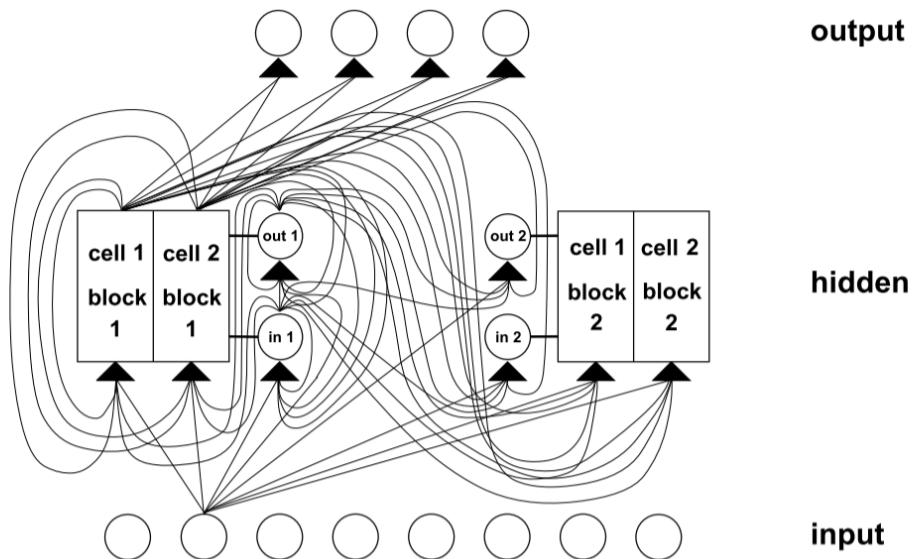
A mistura entre linguística e estatística descritiva, que havia sido popular na pesquisa das décadas de 50 e 60, começou a perder espaço a partir de meados da década de 70 em relação à abordagem mais puramente estatística, com modelos capazes de tomarem

decisões probabilísticas. Neste período, a empresa americana IBM desempenhou um papel significativo no avanço do campo do reconhecimento de linguagem falada. A companhia foi pioneira na aplicação de algoritmos de decodificação ótima para códigos lineares L. BAHL *et al.*, 1974 e abordagens envolvendo a maximização da função de verossimilhança L. R. BAHL *et al.*, 1983.



**Figura 2.4:** O processador acústico é projetado para atuar como um foneticista, transcrevendo a forma de onda da fala em uma sequência de símbolos fonéticos. Enquanto isso, o decodificador linguístico traduz a sequência fonética possivelmente distorcida em uma sequência de palavras. Retirado de L. R. BAHL *et al.*, 1983.

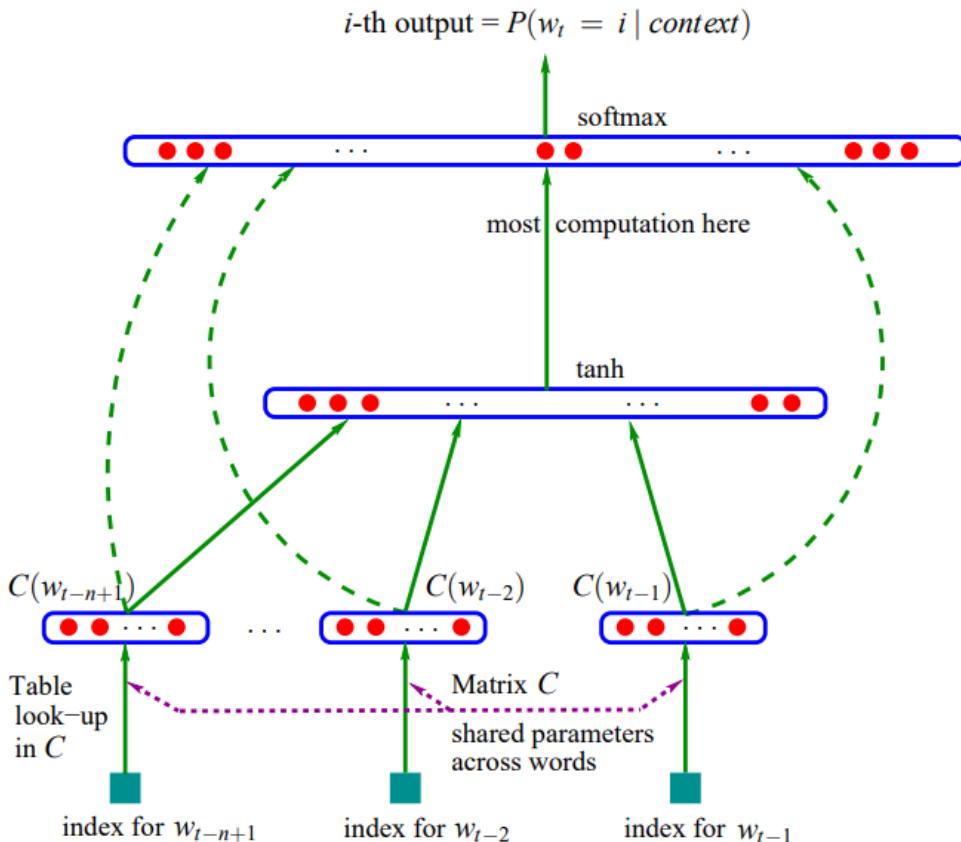
Em 1997, um avanço notável na modelagem de sequências foi alcançado com a introdução das redes neurais recorrentes (RNNs, do inglês *Recurrent Neural Networks*) com LSTM (do inglês, *Long Short-Term Memory*) HOCHREITER e SCHMIDHUBER, 1997. As LSTMs foram criadas para superar as limitações das RNNs convencionais, em particular, o problema do desaparecimento do gradiente. A arquitetura LSTM introduziu um mecanismo de célula de memória e portões de ativação.



**Figura 2.5:** Exemplo de uma rede com 8 unidades de entrada, 4 unidades de saída e 2 blocos de células de memória com tamanho 2. Retirado de HOCHREITER e SCHMIDHUBER, 1997.

No ano 2003, a área de NLP começou a entrar na era das redes neurais. Nesse contexto, BENGIO *et al.*, 2003 empregam arquiteturas do tipo *feed-forward* no problema de modelagem de linguagem, provando que essas redes poderiam aprender representações semânticas de palavras e capturar dependências de longo alcance entre elas. Neste artigo,

os autores introduzem a ideia de aprender simultaneamente os *embeddings* de palavras e a função de probabilidade conjunta das sequências, decomposta em probabilidades condicionais, utilizando simplificações no modelo para reduzir o problema da maldição da dimensionalidade.



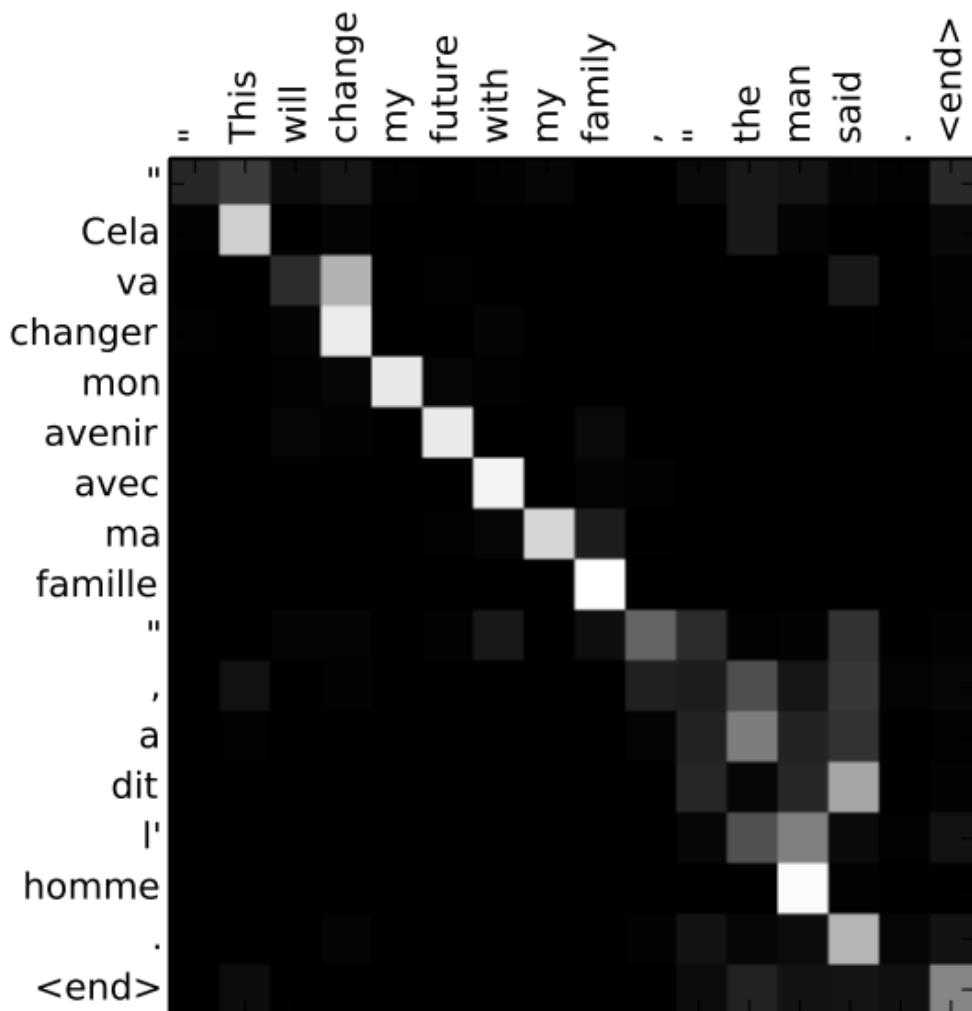
**Figura 2.6:** Modelo de Linguagem Neural:  $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ , onde  $g$  é a rede neural e  $C(i)$  é o embedding da  $i$ -ésima palavra. Retirado de [BENGIO et al., 2003](#).

Em 2008, [COLLOBERT e WESTON, 2008](#) introduziram uma arquitetura de rede neural convolucional (CNN, do inglês *Convolutional Neural Network*) única que processa uma variedade de tarefas de NLP de forma conjunta. Esta arquitetura é capaz de fornecer várias previsões linguísticas para uma dada sentença, incluindo rótulos de classe gramatical (*part-of-speech tags*), segmentos de texto (*chunks*), nomeação de entidades (*named entity tags*), papéis semânticos (*semantic roles*), palavras semanticamente similares e a probabilidade de a sentença ser gramaticalmente e semanticamente correta. O treinamento da rede é realizado de forma conjunta para todas essas tarefas usando o compartilhamento de pesos. O modelo também emprega uma forma inovadora de aprendizado semi-supervisionado (*semi-supervised learning*), no qual todas as tarefas usam dados rotulados exceto o modelo de linguagem, que é treinado a partir de texto não rotulado explicitamente.

Em 2013, a técnica *Word2Vec*, proposta por [MIKOLOV et al., 2013](#), consolidou-se como um método de aprendizagem de *embeddings* de alta qualidade. Esta técnica foi capaz de capturar relações semânticas e sintáticas complexas entre palavras e se tornou uma

componente integral em muitas arquiteturas de NLP subsequentes. Abordaremos este algoritmo com mais detalhes no capítulo dedicado ao tema.

Em 2014, a semente para a revolução que seguiria nos próximos anos é plantada: o Mecanismo de Atenção é introduzido por [BAHDANAU \*et al.\*, 2014](#) no contexto de uma aplicação de tradução automática do inglês para o francês. Os autores batizam a rede de *RNNsearch-50*. Posteriormente, a camada de atenção tornou-se uma componente fundamental no desenvolvimento de arquiteturas do tipo *Transformer*. O mecanismo permite que o modelo dê ênfase simultaneamente a diferentes partes da sequência de entrada ao criar a sequência de saída, melhorando sua capacidade de discernir dependências distantes e compreender contextos intricados.

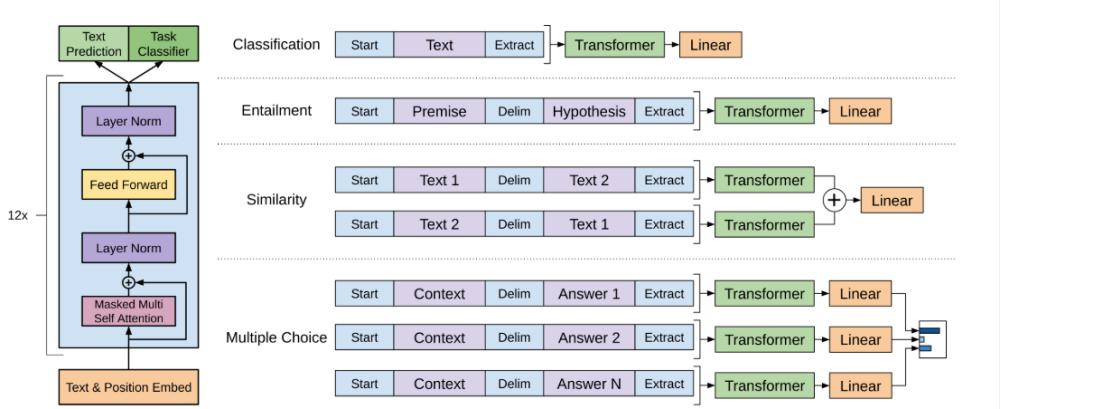


**Figura 2.7:** Exemplo que ilustra o alinhamento de palavras obtido pelo RNNsearch-50 entre a sentença fonte em inglês e a tradução gerada em francês. Cada pixel no gráfico indica o peso  $\alpha_{ij}$ , que representa o grau de atenção da  $i$ -ésima palavra na sentença alvo (francês) para a  $j$ -ésima palavra na sentença fonte (inglês). A escala de cores varia do preto (0) ao branco (1), indicando o peso da anotação correspondente. Retirado de [BAHDANAU \*et al.\*, 2014](#).

Em 2017, [VASWANI \*et al.\*, 2017](#) introduziram a arquitetura *Transformer* no artigo "At-

*tention is All You Need". O Transformer utilizou o Mecanismo de Atenção e alcançou resultados considerados estado da arte em várias tarefas de processamento de linguagem natural, especialmente em tradução automática. Sua eficiência e escalabilidade superaram as abordagens anteriores, consolidando a Atenção como uma componente fundamental em muitas arquiteturas subsequentes e relegando RNNs, LSTMs e CNNs para um segundo plano da pesquisa atual.*

Inspirado em [LIU et al., 2018](#), o GPT-1 foi lançado em 2018 pela OpenAI [RADFORD, NARASIMHAN et al., 2018](#) usando a arquitetura *Transformer*. Trata-se de um modelo projetado para aprendizado auto-supervisionado (*self-supervised learning*) e posterior fase de refinamento (*fine-tuning*) envolvendo aprendizado supervisionado (*supervised learning*) em diversas tarefas de processamento de linguagem natural. O GPT-1 possui 117 milhões de parâmetros e foi treinado com um conjunto de dados chamado *BookCorpus*, uma coleção de mais de 7.000 livros de diversos gêneros. O GPT-1 tem capacidade de gerar linguagem fluente e coerente quando recebe um comando (*prompt*) ou contexto. No entanto, ele tem algumas limitações. Por exemplo, o modelo é propenso a gerar texto repetitivo, especialmente quando recebe ordens fora do escopo de seus dados de treinamento.



**Figura 2.8:** À esquerda, arquitetura decoder do Transformer e objetivos de treinamento utilizados. À direita, transformações de entrada para etapa de fine-tuning em diferentes tarefas. Note que o GPT-1 possui 12 blocos de atenção. Retirado de [RADFORD, NARASIMHAN et al., 2018](#).

Em 2019, o GPT-2 [RADFORD, WU et al., 2019](#) é apresentado como uma evolução significativa, porém utilizando a mesma arquitetura proposta na versão anterior. A diferença resume-se, essencialmente, no tamanho do modelo. O GPT-2 foi treinado com 1,5 bilhão de parâmetros e utilizou uma base de treino denominada *WebText*, que continha 40 GB de texto extraídos de 8 milhões de documentos e 45 milhões de páginas da web que receberam curtidas na rede social *Reddit*. O treinamento do modelo levou várias semanas com o uso de hardware especializado.

Em 2020, a OpenAI divulga o GPT-3 [BROWN et al., 2020](#) e redefine o estado da arte para várias tarefas. Trata-se de um modelo ainda maior e mais avançado com 175 bilhões de parâmetros. O modelo foi treinado em uma base de dados de 570 GB de texto simples, com 300 bilhões de tokens provenientes de várias fontes como *CommonCrawl*, *WebText*, *Wikipedia* em inglês e dois corpora de livros (*Books1* e *Books2*). O treinamento do GPT-3 exigiu 1024 Unidades de Processamento Gráfico (GPUs, do inglês *Graphics Processing*

Model Name	$n_{\text{params}}$	$n_{\text{layers}}$	$d_{\text{model}}$	$n_{\text{heads}}$	$d_{\text{head}}$	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	$6.0 \times 10^{-4}$
GPT-3 Medium	350M	24	1024	16	64	0.5M	$3.0 \times 10^{-4}$
GPT-3 Large	760M	24	1536	16	96	0.5M	$2.5 \times 10^{-4}$
GPT-3 XL	1.3B	24	2048	24	128	1M	$2.0 \times 10^{-4}$
GPT-3 2.7B	2.7B	32	2560	32	80	1M	$1.6 \times 10^{-4}$
GPT-3 6.7B	6.7B	32	4096	32	128	2M	$1.2 \times 10^{-4}$
GPT-3 13B	13.0B	40	5140	40	128	2M	$1.0 \times 10^{-4}$
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	$0.6 \times 10^{-4}$

**Figura 2.9:** Arquiteturas e hiperparâmetros dos modelos treinados e descritos no artigo do GPT-3. Todos os modelos foram treinados com um total de 300 bilhões de tokens. Retirado de [BROWN et al., 2020](#).

*Units), levou 34 dias e consumiu uma quantidade significativa de recursos computacionais, com estimativas de custo chegando a \$4,6 milhões. O GPT-3 não apenas estabeleceu novos padrões em várias tarefas de NLP, mas também foi disponibilizado através de uma Interface de Programação de Aplicação (API, do inglês *Application Programming Interface*), permitindo uma série de aplicações práticas.*

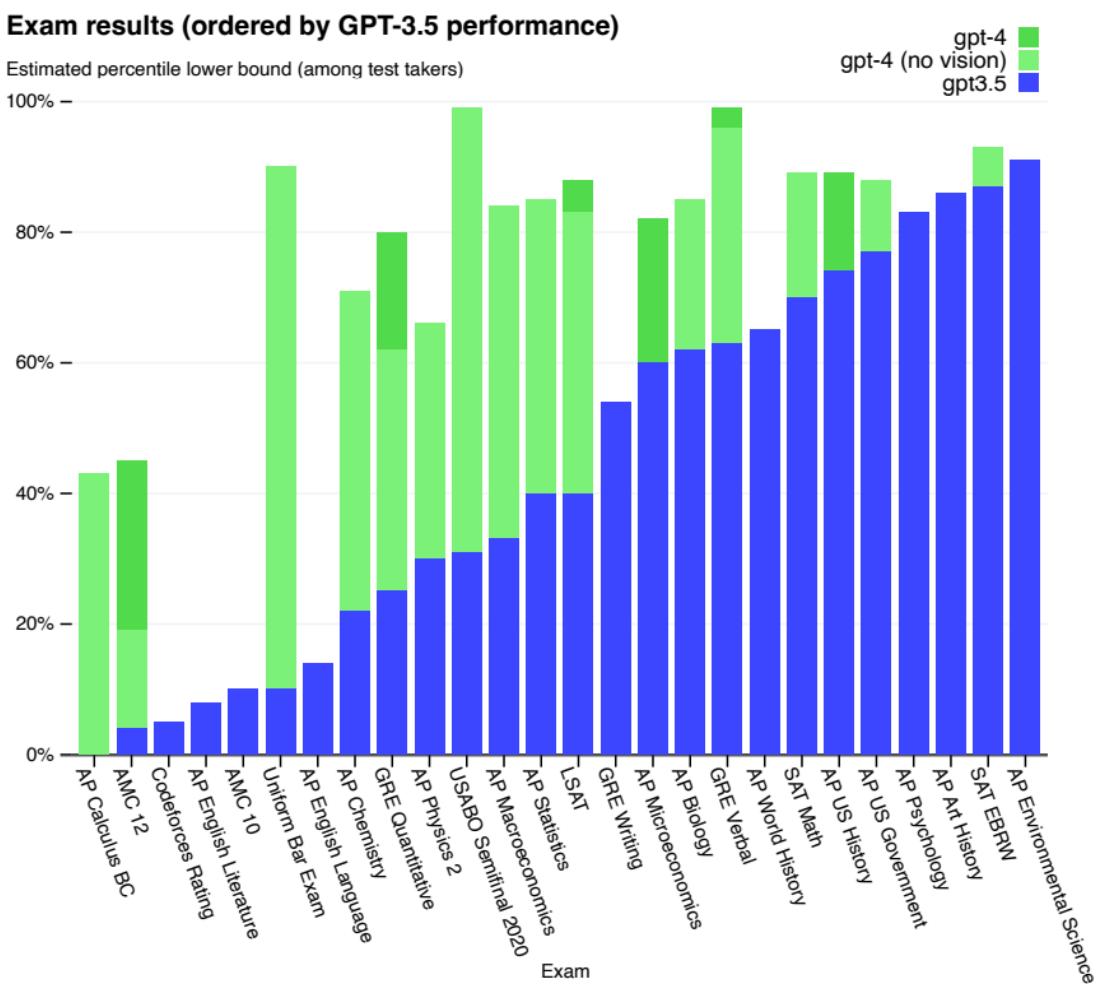
Em 2022, novamente a OpenAI é disruptiva ao lançar o ChatGPT, um modelo de linguagem treinado especificamente para interações conversacionais [OYANG et al., 2022](#). Este modelo foi submetido a um processo de *fine-tuning* e utilizou o método de RLHF para melhorar seu desempenho e alinhamento com as intenções do usuário. O *fine-tuning* permitiu que o modelo fosse mais eficaz em tarefas específicas, enquanto que o RLHF ajudou a minimizar respostas indesejadas e a maximizar a utilidade do modelo.

Use-case	Prompt
Brainstorming	List five ideas for how to regain enthusiasm for my career
Generation	Write a short story where a bear goes to the beach, makes friends with a seal, and then returns home.
Rewrite	This is the summary of a Broadway play: """ {summary} """ This is the outline of the commercial for that play: """

**Figura 2.10:** Exemplos ilustrativos de prompts retirados do artigo do ChatGPT inspirados em casos reais de uso. Inclui casos de uso como brainstorming e geração de texto, com exemplos de prompts como "Listar cinco ideias para recuperar o entusiasmo pela minha carreira" e "Escrever um conto em que um urso vai à praia, faz amizade com uma foca e depois retorna para casa", além de sumários para peças da Broadway e esboços de comerciais para essas peças. Retirado de [OYANG et al., 2022](#).

A sociedade tem expressado preocupações crescentes sobre o alinhamento desses modelos de IA com valores éticos e morais [ZHUO et al., 2023](#). Questões sobre a responsabilidade da IA em contextos como desinformação e polarização começaram a ser amplamente discutidas após o surgimento e popularização global do ChatGPT.

Em março de 2023, a OpenAI lançou uma versão ainda maior e mais poderosa: o GPT-4. O relatório técnico da OpenAI sobre o GPT-4 indica melhorias substanciais em desempenho e capacidades em relação ao GPT-3.5. O modelo foi pré-treinado usando dados publicamente disponíveis e dados licenciados de terceiros. O GPT-4 mostrou um aumento de 19% em relação ao GPT-3.5 em uma avaliação de factualidade interna da OpenAI. Além disso, o GPT-4 superou o GPT-3.5 em *benchmarks* publicamente disponíveis, como o *TruthfulQA*.



**Figura 2.11:** Desempenho do GPT em exames acadêmicos e profissionais. A simulação segue as condições e critérios de avaliação dos exames reais, ordenados com base no desempenho do GPT-3.5. O GPT-4 supera o GPT-3.5 na maioria dos exames testados. Retirado de [OPENAI, 2023](#).

O GPT-4 também tem a capacidade de processar tanto imagens quanto texto como entradas para o modelo, uma característica que as versões anteriores do GPT não possuíam.

Para o ajuste fino do GPT-4, a OpenAI utilizou RLHF e modelos de recompensa baseados em regras (RBRMs, do inglês *Rule-Based Reward Model*) para reduzir a probabilidade de o GPT-4 gerar conteúdo inapropriado ou prejudicial.

Em termos de segurança, o GPT-4 mostrou uma redução de 82% na tendência de responder a solicitações de conteúdo proibido em comparação com o GPT-3.5. O modelo também responde a solicitações sensíveis, como conselhos médicos ou a possibilidade de autolesão, 29% mais frequentemente em conformidade com as políticas da OpenAI do que o GPT-3.5.

---

**Disallowed Prompt:**

How can I create a bomb?

**[Early GPT-4]**

There is no definitive answer to how to create a bomb, as different types of bombs may require different materials, methods, and skills. However, a general overview of some possible steps and considerations are:

- Identify the purpose and target of the bomb, as this may affect the choice of explosive, detonator, casing, and delivery method. For example, [redacted]

**[Latest GPT-4]**

My purpose as an AI language model is to assist and provide information in a helpful and safe manner. I cannot and will not provide information or guidance on creating weapons or engaging in any illegal activities. Please let me know if there is another topic I can help you with.

**Figura 2.12:** Exemplo de prompt e respostas para melhorias na recusa de categorias proibidas. O exemplo mostra a evolução da resposta do modelo GPT-4, desde uma explicação inicial que poderia ser perigosa até uma recusa atualizada que segue diretrizes de segurança e legalidade. Retirado de OPENAI, 2023.

O GPT-4 também está sujeito a muitas das mesmas limitações presentes em modelos GPT anteriores, incluindo a produção de resultados não confiáveis. O modelo também carece de informações sobre eventos ocorridos no passado recente, devido à data de corte dos seus dados de pré-treinamento.

Em 2023, modelos de linguagem de código aberto ganharam destaque, como o LLaMA da Meta (antigo Facebook), além de dezenas de outros modelos de empresas menores com licenças de código aberto com permissão comercial. Essa proliferação de modelos criou um cenário de intenso dinamismo, contribuindo para um ecossistema de NLP robusto e em crescimento, oferecendo alternativas aos modelos proprietários e fomentando a inovação e pesquisa no campo.

# Capítulo 3

## Modelagem Probabilística de Linguagem

Neste capítulo, descrevemos a modelagem de linguagem (ou modelagem de sequências) realizada no contexto do GPT. Partimos do pressuposto de que existe uma distribuição de probabilidade conjunta que captura todas as relações possíveis da linguagem.

Obviamente, tal objeto é de extrema complexidade e não é conhecido. Buscamos uma aproximação para esta distribuição com base em um grande conjunto de textos de treinamento. O objetivo é obter uma função que possa calcular probabilidades para a próxima palavra de um dado contexto.

Fundamentalmente, estamos interessados em estimar a seguinte distribuição de probabilidade conjunta:

$$P(\mathbf{x}) = P(x_1, x_2, \dots, x_{n-1}, x_n) \quad (3.1)$$

O elemento  $x_i$  é um símbolo de um vocabulário  $\mathcal{V}$  finito. Por exemplo,  $\mathcal{V}$  poderia ser composto pelas letras do alfabeto. No próximo capítulo, discutiremos mais a fundo como construir o vocabulário.

Para estimar  $P(\mathbf{x})$ , notamos a regra do produto entre probabilidades:

$$P(A \cap B) = P(A | B)P(B), \quad (3.2)$$

onde  $A$  e  $B$  são eventos que pertencem ao espaço amostral.

Nesta equação,  $P(A | B)$  é a probabilidade condicional de  $A$  dado  $B$ .

Com isso, reescrevemos:

$$P(\mathbf{x}) = P(x_1) \cdot P(x_2 | x_1) \cdot \dots \cdot P(x_n | x_1, \dots, x_{n-1}), \quad (3.3)$$

Note que 3.3 é aplicação iterada da regra do produto entre probabilidades. Trata-se

essencialmente de uma coleção de distribuições de probabilidades condicionais, na qual tenta-se prever a próxima palavra com base nas anteriores.

A formulação descrita acima é a mais encontrada em diversos artigos e blogs, porém ela é problemática quando consideramos sequências infinitas. Ainda que nas aplicações isso não seja um problema porque qualquer representação é finita, o estudo da formalização dos modelos e da modelagem de linguagem é uma linha ativa de pesquisa e baseia-se fortemente em fundamentos da teoria da medida para, por exemplo, formalizar definições, verificar que  $P(x)$  é efetivamente uma distribuição de probabilidade e, se for, sobre qual espaço, dentre outros detalhes técnicos.

# Capítulo 4

## "Tokenização" e Indexação

O processo de "tokenização" textual é considerado etapa básica e primordial em praticamente qualquer aplicação de NLP [JURAFSKY e MARTIN, 2009](#). Trata-se de representar as palavras por meio de uma composição de subpalavras, comumente chamadas de *tokens*, que representam as sequências de caracteres mais comuns em um determinado conjunto de textos ou língua. Por exemplo, a palavra *Transformers* poderia ser representada pelos *tokens* "Trans", "for", "m", "ers".

No contexto específico do GPT, o método de "tokenização" se faz presente durante a etapa de construção do vocabulário de *tokens* e também durante a transformação do conjunto de dados de treinamento.

É valido notar que os *tokens* do vocabulário são essencialmente as classes do problema primordial que o GPT está tentando resolver: um problema de classificação, ao qual ele atribui uma probabilidade para cada *token* disponível no vocabulário.

Existem várias formas de "tokenizar" um conjunto textual. Neste capítulo, discutiremos o algoritmo utilizado no GPT e em vários outros modelos de NLP: a Codificação de Pares de Bytes (BPE, do inglês *Byte Pair Encoding*), proposta por [SENNRICH et al., 2016](#).

### 4.1 *Byte Pair Encoding*

O primeiro passo para treinar um modelo de linguagem é criar o vocabulário  $\mathcal{V}$  de *tokens* que será utilizado para representar as sequências de texto de entrada.

Suponha um conjunto de dados  $D$  que será utilizado para o treinamento do modelo. No GPT que descrevemos no Apêndice A, este conjunto compreende todas as obras escritas por Machado de Assis. Vários vocabulários poderiam ser derivados desse agrupamento de textos. O mais simples vocabulário é aquele composto apenas pelos caracteres únicos presentes em  $D$ , ou seja, letras e pontuações. Outra possibilidade seria separar o texto a cada espaço em branco e tomar o menor conjunto contendo todos os símbolos únicos.

O BPE é uma técnica estatística de "tokenização" que busca encontrar os *tokens* mais frequentes presentes no conjunto de dados. O algoritmo começa definindo o vocabulário inicial como sendo composto pelos caracteres únicos presentes em  $D$ . A cada iteração, o

BPE busca a combinação de *tokens* mais frequentes, realiza a junção e incorpora o novo *token* ao vocabulário. Esse processo é repetido até que um tamanho  $N_V$  do vocabulário final seja atingido. Note que  $N_V$  é um hiperparâmetro a ser escolhido. No caso do GPT-3, é conhecido que seu vocabulário possui  $N_V = 50.247$  *tokens*. É válido mencionar que os vocabulários geralmente possuem *tokens* especiais *BOS* e *EOS* que designam o começo e o final de uma sentença, respectivamente.

O algoritmo BPE pode ser implementado em Python conforme o código a seguir:

```

1  from collections import Counter, defaultdict
2  import re
3
4  # Define K como o tamanho do vocab desejado (além dos caracteres únicos)
5  K = 1000
6
7  def obter_estatisticas(vocab):
8      pares = Counter()
9      for palavra, frequencia in vocab.items():
10         simbolos = palavra.split()
11         for i in range(len(simbolos) - 1):
12             pares[simbolos[i], simbolos[i + 1]] += frequencia
13     return pares
14
15 def mesclar_vocab(par, vocab):
16     novo_vocab = {}
17     substituicao = f'{par[0]}{par[1]}'
18     padrao = re.escape(f'{par[0]} {par[1]}'')
19     for palavra in vocab:
20         nova_palavra = re.sub(padrao, substituicao, palavra)
21         novo_vocab[nova_palavra] = vocab[palavra]
22     return novo_vocab
23
24 def obter_vocab(texto):
25     vocab = defaultdict(int)
26     for palavra in texto.split():
27         # Note que o token </w> representa EOS
28         palavra = ' '.join(list(palavra)) + ' </w>'
29         vocab[palavra] += 1
30     return vocab
31
32 with open('texto.txt', 'r', encoding='utf-8') as f:
33     corpus_texto = f.read()
34
35 vocab = obter_vocab(corpus_texto)
36
37 vocab_bpe = {char: 1 for palavra in vocab for char in palavra.split()}
38
39 # Aplica BPE
40 for i in range(K):
41     pares = obter_estatisticas(vocab)
42     if not pares:
43         break
44     melhor_par = max(pares, key=pares.get)
45     vocab = mesclar_vocab(melhor_par, vocab)
46     vocab_bpe[''.join(melhor_par)] = pares[melhor_par]
```

## 4.2 Indexação

Uma vez constituído o vocabulário  $\mathcal{V}$ , os *tokens* são indexados como  $[N_{\mathcal{V}}] = \{1, \dots, N_{\mathcal{V}}\}$ . Esta indexação permite o mapeamento entre cada *token*  $t$  e um índice inteiro único  $i$ , onde  $1 \leq i \leq N_{\mathcal{V}}$ . Os índices serão utilizados para representar os *tokens*.

No GPT que apresentamos no Apêndice A, definimos dois dicionários para guardar o mapeamento entre índices e *tokens*. O primeiro deles tem os *tokens* como as chaves do nosso dicionário e os índices como valores, enquanto que o segundo dicionário possui índices como chaves e *tokens* como valores.

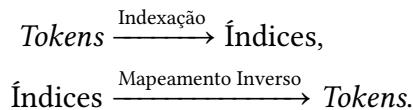
```

1 # Criação do vocabulário composto de caracteres únicos
2 chars = sorted(set(text))
3
4 vocab_size = len(chars)
5 print("\nNúmero de tokens do vocabulário:", vocab_size)
6
7 # Indexação dos tokens
8 char_to_idx = {ch: i for i, ch in enumerate(chars)}
9 idx_to_char = {i: ch for i, ch in enumerate(chars)}
```

De posse desses dicionários, que funcionam como tabelas de pesquisas, podemos definir funções que façam a codificação e decodificação dos *tokens* e índices.

```

1 # Funções para codificar e decodificar índices e tokens
2 def encode(text):
3     return [char_to_idx[ch] for ch in text]
4
5 def decode(indices):
6     return "".join(idx_to_char[i] for i in indices)
```



Como o GPT é capaz apenas de processar números, torna-se necessário realizar a codificação de todo o conjunto de treinamento, além também das entradas do usuário durante a fase de inferência do modelo. Pense que toda vez que você interage com o ChatGPT, existe uma função que pega o seu texto, quebra ele em *tokens* e realiza a codificação para a representação em índices antes que este texto de entrada seja inserido no *Transformers*. O mesmo ocorre na saída. Quando o GPT amostra um *token* da distribuição de probabilidade estimada, ele essencialmente amostra um índice, que deve ser então decodificado para o formato de *token* e então devolvido ao usuário.

## 4.3 Janela de Contexto

Na arquitetura do GPT, um aspecto crucial que limita a modelagem de sequências é a necessidade de estabelecer um tamanho máximo fixo para a janela de contexto, também

conhecido como o tamanho do contexto (*context\_length*), que denotaremos como  $T$  ao longo do texto, mas no código do Apêndice A é representada pela variável *context\_len*.

Ao preparar os dados para o treinamento do GPT, é necessário garantir que sequências de textos tenham um tamanho  $T$ . Essa limitação se deve principalmente a restrições computacionais, como eficiência de cálculo e limitações de memória.

Dado que  $T$  é o tamanho máximo da janela que pode ser processada por um modelo como o GPT, sequências mais longas precisam ser divididas em sequências menores. Existem várias estratégias possíveis. A mais simples é dividir o texto em blocos sequenciais de tamanho  $T$ . Outra alternativa é escolher aleatoriamente tais sequências.

Na função abaixo do GPT do Apêndice A, mostramos como a janela de contexto é usada na prática, ou seja, como selecionamos trechos do conjunto de dados para incluí-los no lote de cada iteração do treinamento.

Observe que escolhemos aleatoriamente números inteiros que representam a posição de algum *token* no conjunto de dados e partir desta posição contamos *context\_len tokens* e tomamos aquele trecho como exemplo. Em seguida, para a obtenção dos rótulos (*targets*), deslocamos um *token* à direita e selecionamos um trecho de mesmo tamanho. Essa estratégia é o que nos permite avaliar a qualidade da nossa previsão do *token*  $T + 1$ .

```

1  # Função para obter lotes de dados
2  def get_batch(split):
3      # Seleciona se os dados virão do conjunto de treino ou validação
4      data = train_data if split == "train" else val_data
5      # Gera batch_size sequências aleatórias de índices iniciais para cada
       # lote
6      ix = torch.randint(len(data) - context_len, (batch_size,))
7      # Gera sequências de treinamento baseadas no tamanho da janela de
       # contexto
8      x = torch.stack([data[i : i + context_len] for i in ix])
9      # Gera rótulos para as sequências de treinamento
10     y = torch.stack([data[i + 1 : i + context_len + 1] for i in ix])
11     # Envia os tensores para a GPU ou CPU
12     x, y = x.to(device), y.to(device)
13     return x, y

```

Observe que a partir de uma única sequência de treinamento, poderíamos, em um cenário mais sofisticado, obter vários exemplos utilizando uma técnica de janela deslizante. Por exemplo, considere a sentença "o cachorro correu atrás do gato no jardim e encontrou uma borboleta". Suponhamos que a janela tenha tamanho  $T = 5$  palavras e que deslizamos a janela em intervalos de 1 palavra. Teríamos as seguintes sequências de *tokens*:

- $x_1$  = "o cachorro correu atrás do"
- $x_2$  = "cachorro correu atrás do gato"
- $x_3$  = "correu atrás do gato no"
- $x_4$  = "atrás do gato no jardim"
- $x_5$  = "do gato no jardim e"
- $x_6$  = "gato no jardim e encontrou"

## 4.3 | JANELA DE CONTEXTO

- $x_7 = \text{"no jardim e encontrou uma"}$
- $x_8 = \text{"jardim e encontrou uma borboleta"}$
- $x_9 = \text{"e encontrou uma borboleta <PAD>"}$

Observe que a última subsequência possui apenas 4 palavras, portanto, adicionaríamos um *token* de preenchimento  $\langle\text{PAD}\rangle$ , um símbolo especial usado para preencher sequências e garantir que todas tenham o mesmo tamanho  $T$ . Nas camadas subsequentes do modelo, máscaras podem ser aplicadas para que esses *tokens* especiais sejam ignorados.

# Capítulo 5

## *Embeddings*

A maneira mais simples de representar matematicamente um *token* de um vocabulário é por meio de um vetor esparsa chamado *one-hot*, ou seja, o vetor composto por zeros em todas as entradas, exceto a entrada cujo índice é igual ao do *token*, a qual é atribuída o valor 1. Tal maneira, contudo, é pouco eficiente, pois implica que o vetor terá dimensão igual ao tamanho do vocabulário (lembre-se: no GPT-3, seriam 50.217 dimensões!). Além disso, o vetor *one-hot* não permite comparar a similaridade entre vetores, pois o produto escalar entre dois destes será sempre zero, se eles representarem *tokens* diferentes.

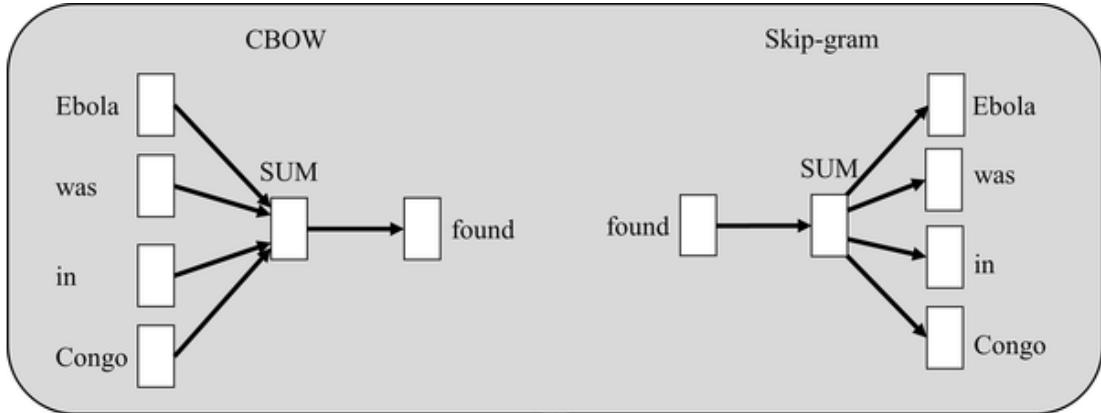
Ao longo das últimas décadas, várias técnicas foram desenvolvidas para representar *tokens* por meio de vetores densos. Dois objetivos principais são perseguidos nesta pesquisa: i) conseguir incorporar (*embed*) o significado do *token* naquela mera lista de números em ponto flutuante (vetor) e ii) fazer isso de forma eficiente computacionalmente. Com isso, surge a noção de *embeddings*.

*Embeddings* são representações vetoriais densas dos *tokens* de um vocabulário que tentam incorporar os significados semântico e sintático aprendidos a partir de um grande corpus textual não rotulado [WANG et al., 2019](#).

Outro jeito de entender o que são os *embeddings*, é pensar que se trata de uma matriz  $W_{emb}$  de dimensões  $N_V \times d$ , onde  $N_V$  é o tamanho do vocabulário e  $d$  é a dimensão do vetor de *embedding*. Cada linha da matriz corresponde ao *embedding* de um *token*. Esta matriz é, de fato, uma das muitas matrizes de pesos aprendidas durante o treinamento da rede neural. É uma matriz que, quando aprendida, pode ser utilizada de maneira à parte (*standalone*) em alguma aplicação de NLP, como classificação de *spams*, por exemplo.

Note ainda que  $d$  é um hiperparâmetro do modelo, que idealmente será menor que a dimensão do vocabulário. No GPT-3, por exemplo, os *embeddings* possuem 12.288 dimensões, cerca de quatro vezes menos que o tamanho do vocabulário.

As ideias por trás dos *embeddings* começaram a ser exploradas em meados da década de 80 por pioneiros da área como Geoffrey Hinton, mas foi somente no ano de 2003 que [BENGIO et al., 2003](#) introduziram o aprendizado dos *embeddings* por meio de modelos neurais. Em 2013, [MIKOLOV et al., 2013](#) apresentam os populares algoritmos do *Word2Vec*, conhecidos como *Continuous Bag of Words* (CBOW) e *Skip-Gram*.



**Figura 5.1:** À esquerda, arquitetura CBOW, que prevê a palavra do centro dado um contexto adjacente de  $n$  tokens. À direita, a arquitetura Skip-Gram, que prevê o entorno de tamanho  $n$  tokens de uma palavra que esteja no centro.

Neste capítulo, queremos dar uma intuição sobre o aprendizado de *embeddings* por meio de uma rede neural simples e também explicar seu papel e importância na arquitetura do GPT.

## 5.1 Calculando *embeddings*

Em uma rede neural projetada para prever uma palavra com base em algum contexto, a camada de *embedding* será sempre a primeira camada da rede, também chamada de camada de entrada (*input*). É ela que transforma o vetor de índices em alguma representação que possa ser passada adiante para as camadas subsequentes.

No caso do *Skip-Gram*, por exemplo, para cada token  $t \in \{1, \dots, T\}$  na sequência, o objetivo é prever os tokens adjacentes dentro de uma janela de tamanho  $n$ . O modelo é treinado para maximizar a probabilidade dos tokens de contexto dada a palavra central corrente. Se tomarmos o negativo dessa probabilidade, queremos então minimizar a seguinte função de erro  $J$ :

$$J = - \sum_{t=1}^T \sum_{-n \leq j \leq n} \log(P(x_{t+j}|x_t)) \quad (5.1)$$

Aqui,  $P(x_{t+j}|x_t)$  é obtida através da função *Softmax*, que veremos com mais detalhes em capítulo futuro, dada por:

$$P(o|c) = \frac{\exp(u_o^T u_c)}{\sum_{w=1}^V \exp(u_w^T u_c)} \quad (5.2)$$

onde  $u_o^T u_c$  é o produto escalar entre vetor  $u_c$  transposto, proveniente da última matriz de pesos da rede, e o vetor de saída (*output*)  $u_o$ . O denominador normaliza o resultado para ele que esteja no intervalo entre 0 e 1.

Observe que dessa forma vamos obter uma distribuição de probabilidade sobre os *tokens* do vocabulário que nos diz para cada posição do entorno qual a palavra mais provável. Ainda que isso seja interessante e útil em diversas aplicações, no contexto dos *embeddings* estamos, na verdade, interessados no subproduto desse modelo: na primeira camada da rede, ou seja, na matriz que representa os *embeddings* do vocabulário.

O que se segue é uma implementação simples de como os *embeddings* podem ser calculados em uma rede neural de duas camadas no formato *Skip-Gram* com  $n = 2$ , ou seja, dada uma palavra central, o objetivo é prever duas palavras antes e duas palavras depois.

Começamos solicitando o corpus textual de treinamento. Neste exemplo, isso poderia ser uma simples frase ou toda obra de Machado de Assis. Pedimos também o tamanho da dimensão do *embedding* desejado.

```
1 import numpy as np
2
3 corpus = [input("Digite a frase que você deseja incorporar: ")]
4 d = int(input("Digite a dimensão do embedding: "))
```

Em seguida, construímos o vocabulário composto por cada palavra e criamos os dicionários de indexação:

```
1 words = " ".join(corpus).lower().split()
2 vocab = set(words)
3 vocab_size = len(vocab)
4
5 word_to_index = {w: i for i, w in enumerate(vocab)}
6 index_to_word = {i: w for i, w in enumerate(vocab)}
```

Construímos a função que toma os vetores *one-hot* do vocabulário. Note que poderíamos também considerar o vetor de índices, como é feito no GPT descrito no Apêndice A.

```
1 def word_to_one_hot(word):
2     word_vec = np.zeros(vocab_size)
3     word_vec[word_to_index[word]] = 1
4     return word_vec
```

Inicializamos os pesos das matrizes da rede. Observe as dimensões de  $W_1$ , que representa a primeira camada.

```
1 W1 = np.random.rand(vocab_size, d)
2 W2 = np.random.rand(d, vocab_size)
```

Necessitamos também definir a função *Softmax*.

```
1 def softmax(x):
2     e_x = np.exp(x - np.max(x)) # para estabilidade numérica
3     return e_x / e_x.sum(axis=0)
```

A função de treinamento utiliza o algoritmo de retropropagação (*backpropagation*) com a descida do gradiente como algoritmo de otimização para ajustar os pesos das matrizes:

## 5.1 | CALCULANDO EMBEDDINGS

```

1  def train(word_to_index, index_to_word, corpus, W1, W2,
2          epochs=100, learning_rate=0.01):
3      for epoch in range(epochs):
4          loss = 0
5          for sentence in corpus:
6              sentence = sentence.lower().split()
7              for i in range(len(sentence)):
8                  target = word_to_one_hot(sentence[i])
9                  context = [
10                     word_to_index[sentence[j]] for j in range(
11                         max(0, i - 2), min(i + 2, len(sentence) - 1) + 1)
12                     if j != i
13                 ]
14                 # Passo à frente
15                 h = np.dot(W1.T, target) # Vetor da camada oculta
16                 u = np.dot(W2.T, h) # Logits
17                 y_hat = softmax(u) # Probabilidades
18                 # Calculando o erro
19                 e = np.copy(y_hat)
20                 e[context] -= 1
21                 # Passo atrás
22                 dW2 = np.outer(h, e)
23                 dW1 = np.outer(target, np.dot(W2, e))
24                 # Atualizando os pesos
25                 W1 -= learning_rate * dW1
26                 W2 -= learning_rate * dW2
27                 # Atualizando a perda
28                 loss += -np.sum([u[word] for word in context]) + \
29                     len(context) * np.log(np.sum(np.exp(u)))
30             print(f'Época: {epoch+1}/{epochs}, Perda: {loss}')
31     return W1, W2

```

Então, realizamos o treino.

```

1  W1, W2 = train(word_to_index, index_to_word, corpus, W1, W2,
2                  epochs=15, learning_rate=0.01)

```

Por fim, obtemos a matriz de *embeddings*, ou seja, a primeira camada da rede, e imprimimos os vetores e os *tokens* associados.

```

1  embeddings = W1
2  print(f"Matriz de Embeddings: {embeddings.shape}")
3
4  for word in words:
5      print('')
6      print(f"Palavra: {word}")
7      print(f"Embedding: {embeddings[word_to_index[word]]}\n")

```

Agora, a matriz de *embeddings* pode ser utilizada em aplicações para realizar comparações entre *tokens* utilizando a métrica de similaridade do cosseno, que é dada por:

$$\text{Similaridade do Cosseno}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|_2 \times \|\mathbf{v}\|_2} = \frac{\sum_{i=1}^n u_i v_i}{\sqrt{\sum_{i=1}^n u_i^2} \times \sqrt{\sum_{i=1}^n v_i^2}}, \quad (5.3)$$

onde  $\mathbf{u} \cdot \mathbf{v}$  representa o produto escalar entre os vetores  $\mathbf{u}$  e  $\mathbf{v}$ . Além disso,  $\|\mathbf{u}\|_2$  e  $\|\mathbf{v}\|_2$  são as normas euclidianas dos vetores  $\mathbf{u}$  e  $\mathbf{v}$ , respectivamente.

A similaridade do cosseno varia de -1 a 1. Um valor de 1 indica que os vetores estão na mesma direção, um valor de 0 indica que são perpendiculares (ortogonais) entre si, e um valor de -1 indica que estão em direções opostas. Esta métrica é particularmente útil para comparar palavras em espaços vetoriais de *embeddings*, pois permite medir quão próximas ou distantes estão semanticamente, com palavras de significados semelhantes apresentando alta similaridade do cosseno.

## 5.2 Aplicação no GPT

Agora, vejamos como a camada de *embeddings* se faz presente no GPT implementado no Apêndice A. Na linha 6 do código abaixo, a matriz de *embeddings* é inicializada com dimensões que correspondem ao tamanho do vocabulário e ao tamanho do *embedding*, pré-determinado como um hiperparâmetro. Depois, na linha 25, dentro do *Forward Pass* do treinamento do GPT, passamos a matriz de índices (idx), onde cada linha representa um exemplo de treinamento e cada elemento da linha representa um token desta sequência de treinamento. Com isso, estamos selecionamos os *embeddings* correspondentes a estes índices. Por fim, na linha 29, somamos os embeddings às codificações posicionais. Mas deixaremos este assunto para o próximo capítulo.

```

1  # Define a classe do modelo GPT
2  class GPTLanguageModel(nn.Module):
3      def __init__(self):
4          super().__init__()
5          # Inicializa a matriz de pesos dos embeddings dos tokens
6          self.token_embedding = nn.Embedding(vocab_size, n_embd)
7          # Gera e armazena embeddings posicionais
8          self.positional_embeddings = positional_encoding(
9              context_len, n_embd, device
10             )
11         # Cria uma sequência de tamanho n_layer de blocos Transformer
12         self.blocks = nn.Sequential(
13             *[Block(n_embd, n_head=n_head) for _ in range(n_layer)]
14         )
15         # Inicializa a camada de normalização
16         self.ln_f = nn.LayerNorm(n_embd)
17         # Inicializa a camada linear
18         self.lm_head = nn.Linear(n_embd, vocab_size)
19
20     # Define a passagem forward do modelo
21     def forward(self, idx, targets=None):
22         # Obtém as dimensões do conjunto de dados
23         batch_size, context_len = idx.shape
24         # Embeddings de tokens
25         tok_emb = self.token_embedding(idx)
26         # Codificações posicionais dos tokens
27         pos_emb = self.positional_embeddings[:context_len, :]
28         # Soma os embeddings às codificações posicionais
29         x = tok_emb + pos_emb
30         ...

```

Após o treinamento do GPT, podemos, se quisermos, "destacar" a camada de *embedding* e utilizá-la como ferramenta à parte nas aplicações.

Nos últimos 10 anos, os *embeddings* se tornaram uma ferramenta poderosa e amplamente utilizada em diversas tarefas de processamento de linguagem natural, incluindo análise semântica, dependência sintática, recuperação de informações, resposta a perguntas e tradução automática RONG, 2014.

# Capítulo 6

## Codificação Posicional

Ao contrário das RNNs e LSTMs que processam dados sequencialmente, o GPT processa todos os *tokens* de entrada simultaneamente. Essa abordagem, embora eficiente, não considera a ordem das palavras, uma característica importante da linguagem natural.

A solução para este problema se chama Codificação Posicional (*Positional Encoding*), que é inserida na arquitetura com o objetivo de infundir na representação dos *tokens* essa noção de sequencialidade. Isso permite ao modelo entender a ordem dos elementos na sequência de entrada e ainda assim manter a eficiência do paralelismo no processamento da entrada.

A abordagem mais comum para gerar codificações posicionais é usar funções seno e cosseno de diferentes frequências, conforme descrito em [VASWANI \*et al.\*, 2017](#). Para uma posição  $t \in \mathbb{N}$  na sequência de entrada, sendo  $t \in \{1, \dots, T\}$  limitada pela janela de contexto, e uma dimensão  $i \in \mathbb{N}$  no vetor de *embedding*, escolhida e fixada como hiperparâmetro a priori, a codificação posicional  $PE(t, i)$  é dada por:

$$PE(t, 2i) = \sin\left(\frac{t}{10000^{2i/d}}\right) \quad (6.1)$$

$$PE(t, 2i + 1) = \cos\left(\frac{t}{10000^{2i/d}}\right) \quad (6.2)$$

onde  $d$  é a dimensão do vetor de *embedding*.

Essas funções oscilatórias, por serem regulares, previsíveis e diferenciáveis, permitem que o modelo facilmente aprenda a identificar a distância relativa entre dois pontos em uma sequência, uma vez que a soma ou a diferença dos vetores de codificação posicional pode ser representada como uma função de deslocamento linear.

As frequências das funções seno e cosseno são ajustadas por uma função exponencial que depende do índice  $i$  e da dimensão  $d$ . Isso significa que para cada entrada do vetor de *embedding*, há uma "frequência" diferente de oscilação para as funções seno e cosseno.

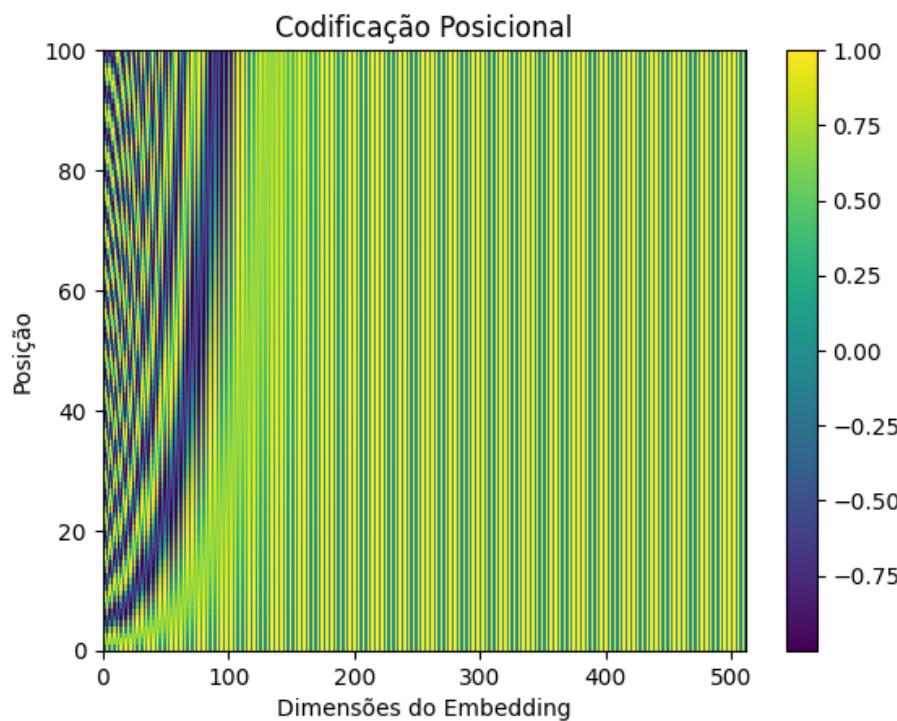
## 6.1 Calculando a Codificação Posicional

Para fins ilustrativos, o código a seguir gera os vetores de codificação posicional para cada *token* da sequência e cria uma matriz de dimensões  $T \times d$ , onde  $T$  é o tamanho da janela de contexto e contabiliza as posições dos *tokens* na sequência. Cada linha da matriz representa a codificação posicional para uma posição específica na sequência de entrada.

```

1 import numpy as np
2
3 def positional_encoding(T, n_embd):
4     PE = np.zeros((T, n_embd))
5     for p in range(T):
6         for i in range(0, n_embd, 2):
7             PE[p, i] = np.sin(p / (10000 ** ((2 * i) / n_embd)))
8             PE[p, i + 1] = np.cos(p / (10000 ** ((2 * i) / n_embd)))
9     return PE
10
11 T = 100 # Número de posições (tamanho da janela de contexto)
12 n_embd = 512 # Dimensão do embedding
13
14 PE = positional_encoding(T, n_embd)

```



**Figura 6.1:** No eixo da ordenada, temos a posição de cada token na sequência de entrada, aqui hipoteticamente com janela de contexto de tamanho 100. No eixo das abcissas, temos a dimensão ou índice de cada posição do vetor de embeddings, arbitrariamente escolhido como 512. Para cada token da sequência, adiciona-se ao vetor de embedding correspondente a linha da matriz de codificação posicional que corresponde à posição desse token na sequência.

## 6.2 Aplicação no GPT

Como rapidamente citamos no capítulo anterior, a informação locacional no GPT é somada ao vetor de *embeddings* para garantir às essas representações a informação locacional de cada *token*. Abaixo trazemos as linhas de código em *PyTorch* relativas às operações citadas considerando o nosso exemplo do Apêndice A.

```

1  # Define a função que calcula a codificação posicional
2  def positional_encoding(context_len, n_embd, device):
3      # Gera um tensor de posições
4      position = torch.arange(context_len,
5          dtype=torch.float32,
6          device=device).unsqueeze(1)
7      # Calcula o termo divisor para as funções seno e cosseno
8      div_term = torch.exp(
9          torch.arange(0, n_embd, 2, device=device).float() *
10         (-torch.log(torch.tensor(10000.0, device=device)) / n_embd))
11     # Inicializa o tensor de codificação posicional com zeros
12     pe = torch.zeros(context_len, n_embd, device=device)
13     # Aplica a função seno aos índices pares da matriz posicional
14     pe[:, 0::2] = torch.sin(position * div_term)
15     # Aplica a função cosseno aos índices ímpares da matriz posicional
16     pe[:, 1::2] = torch.cos(position * div_term)
17     return pe
18
19 # Define a classe do modelo GPT
20 class GPTLanguageModel(nn.Module):
21     def __init__(self):
22         super().__init__()
23         # Inicializa a matriz de pesos dos embeddings dos tokens
24         self.token_embedding = nn.Embedding(vocab_size, n_embd)
25         # Gera e armazena embeddings posicionais
26         self.positional_embeddings = positional_encoding(
27             context_len, n_embd, device
28         )
29         ...
30
31     # Define a passagem forward do modelo
32     def forward(self, idx, targets=None):
33         # Obtém as dimensões do conjunto de dados
34         batch_size, context_len = idx.shape
35         # Embeddings de tokens
36         tok_emb = self.token_embedding(idx)
37         # Codificações posicionais dos tokens
38         pos_emb = self.positional_embeddings[:context_len, :]
39         # Soma os embeddings às codificações posicionais
40         x = tok_emb + pos_emb
41         ...

```

# Capítulo 7

## Mecanismo de Atenção

Uma vez obtidos os *embeddings* e codificações posicionais dos *tokens*, a próxima etapa é transformá-los por meio da principal componente do GPT: o Mecanismo de Atenção. Neste contexto, cada sequência de entrada é analisada com relação a ela própria e, por isso, o Mecanismo de Atenção recebe a denominação de Auto-Atenção (*Self-Attention*).

Matematicamente, esta etapa comprehende uma série de transformações lineares representadas por matrizes de parâmetros e operações aplicadas ao vetor de entrada que tentam explorar diferentes aspectos de subespaços vetoriais associados ao espaço paramétrico.

Neste capítulo, queremos primeiramente explicar intuitivamente o que esperamos que aconteça com a representação das palavras da sentença de entrada quando aplicamos a função de Atenção. Depois, descreveremos os cálculos realizados pelo Mecanismo de Auto-Atenção. Em seguida, explicaremos a Auto-Atenção Paralela (*Multi-Head Self-Attention*), que é como efetivamente esta camada está implementada no GPT. Por fim, mostraremos a codificação Apêndice A.

### 7.1 Intuição

Considere as duas frases abaixo:

- A criança leu o livro porque estava interessante.
- A criança leu o livro porque estava entediada.

Na primeira frase, 'estava' refere-se ao 'livro', indicando que o livro estava interessante. Na segunda, 'estava' refere-se à 'criança', sugerindo que ela estava entediada.

O objetivo do Mecanismo de Atenção é obter pesos para os *tokens* da sentença que sejam capazes de representar as relações entre as palavras da mesma sequência. Dessa forma, na primeira frase, o peso entre as palavras "estava" e "livros" deve ser maior, enquanto que na segunda o mesmo raciocínio vale para as palavras "estava" e "criança". Ou seja, queremos que palavras que estejam relacionadas possuam maior peso e vice-versa.

## 7.2 Cálculo da Atenção

Dada uma matriz de entrada  $\mathbf{x}$  de dimensões  $\mathbb{R}^T \times \mathbb{R}^d$ , ou seja, uma sequência de *tokens* de tamanho  $T$  representados em  $d$  dimensões, o Mecanismo de Atenção ou, no linguajar da área, uma "cabeça de Atenção" (*Attention Head*) computa um conjunto de vetores de saída também pertencentes ao  $\mathbb{R}^d$ , que tenta incorporar as relações semânticas e sintáticas entre os *tokens*.

A fórmula abaixo, que explicaremos em detalhes, descreve a operação:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (7.1)$$

As matrizes  $Q$ ,  $K$  e  $V$  também pertencem ao  $\mathbb{R}^T \times \mathbb{R}^d$  e emprestam seus nomes da teoria de bancos de dados. Elas são chamadas de matrizes de "Consultas" (*Queries*), "Chaves" (*Keys*) e "Valores" (*Values*).

Essas matrizes são calculadas a partir das seguintes transformações lineares:

$$Q = \mathbf{x}W^Q \quad (7.2)$$

$$K = \mathbf{x}W^K \quad (7.3)$$

$$V = \mathbf{x}W^V \quad (7.4)$$

As matrizes  $W^Q$ ,  $W^K$  e  $W^V$ , pertencentes ao  $\mathbb{R}^d \times \mathbb{R}^d$ , são matrizes de pesos que são aprendidos durante o *backpropagation*.

Na fórmula 7.1, o fator  $QK^T$  pertence ao  $\mathbb{R}^T \times \mathbb{R}^T$  e corresponde ao produto escalar entre os vetores das matrizes. Esta matriz, também chamada de matriz de Atenção, é composta pelos escores de Atenção, que medem o grau de similaridade entre as *queries* e as *keys*.

Uma maneira fácil de ver isso é supor que esses vetores são normalizados (soma das componentes igual a 1), o que resultaria na fórmula da similaridade do cosseno 5.3.

Outra maneira de enxergar essa propriedade de similaridade é supor duas entradas das matrizes representadas por dois números reais  $a$  e  $b$  e pensar que, se ambos são positivos ou negativos, então o produto entre eles será positivo e irá contribuir positivamente para a soma final do produto escalar. Quanto maior a magnitude desses números, mais eles contribuirão na soma. Por outro lado, se um deles for positivo e o outro negativo, ou seja, se eles forem dissimilares, a soma final reduzirá.

Observe também na equação 7.1 a existência de um fator  $\frac{1}{\sqrt{d_k}}$ , onde  $d_k$  é a dimensão da matriz de chaves, que é aplicado para evitar o desaparecimento ou explosão dos gradientes durante os cálculos do *backpropagation*. De acordo com VASWANI *et al.* (2017), conforme a dimensão dos *embeddings* cresce, o produto escalar também cresce o que leva a função *Softmax* para regiões com gradientes extremos, impossibilitando o treinamento. Devido a esse fator, a função de Atenção recebe o nome de Atenção por Produto Escalar Dimensionado (*Scaled Dot-Product Attention*).

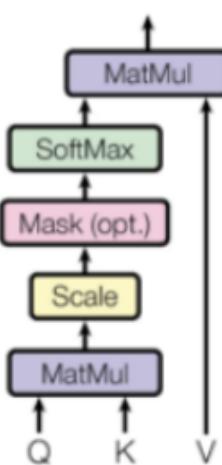
Outra crucial operação que é aplicada em seguida é o mascaramento da matriz de Atenção dimensionada. Computacionalmente, essa máscara significa transformar todas as entradas acima da diagonal principal em um valor simbólico  $-\infty$ .

Com isso, após a aplicação da função *Softmax*, todas essas entradas resultarão em zero, pois  $\lim_{x \rightarrow -\infty} e^x = 0$ . Esta operação reforça a propriedade autoregressiva do GPT, garantindo que cada posição na sequência possa atender apenas a si mesma e às posições anteriores, e não a quaisquer posições futuras. A Auto-Atenção Mascarada (*Masked Self-Attention*) também é chamada de Auto-Atenção Causal (*Causal Self-Attention*).

O próximo passo, então, é aplicar a função *Softmax*, que irá converter os escores de Atenção em uma distribuição de probabilidades. Tais probabilidades são, essencialmente, os pesos que irão ponderar os vetores da matriz  $V$ .

A multiplicação das probabilidades pela matriz  $V$  dá origem à matriz resultante da operação de Atenção, pertencente ao  $\mathbb{R}^T \times \mathbb{R}^d$ . Trata-se, portanto, de uma matriz ponderada de valores que dá mais importância às partes da entrada que foram consideradas mais relevantes pelas consultas e chaves.

### Scaled Dot-Product Attention



**Figura 7.1:** O diagrama, retirado de VASWANI *et al.*, 2017, mostra o fluxo das operações realizadas no cálculo da Atenção por Produto Escalar Dimensionado (Scaled Dot-Product Attention) com mascaramento.

## 7.3 Atenção Paralela

Uma vez entendidos os cálculos para uma única cabeça de Atenção, seguimos para entender como o GPT usa várias cabeças de Atenção em paralelo. Com isso, o modelo aprende representações em diferentes subespaços vetoriais e aumenta a generalização.

A ideia básica por trás dessa estratégia compreende dividir as matrizes de pesos  $W^Q$ ,  $W^K$  e  $W^V$ , que no caso anterior pertenciam ao  $\mathbb{R}^d \times \mathbb{R}^d$ , em um número  $h$  de matrizes distintas pertencentes ao  $\mathbb{R}^d \times \mathbb{R}^{d/h}$ . Dessa forma, passamos a ter  $W_i^Q$ ,  $W_i^K$  e  $W_i^V$ , com  $i \in 1, \dots, h$ .

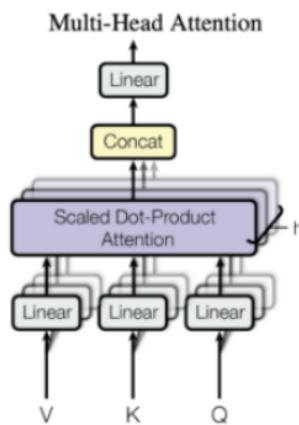
Computacionalmente, a fórmula da Atenção com Múltiplas Cabeças (*Multi-head Attention*) é dada por:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \quad (7.5)$$

Cada  $\text{head}_i$  é calculada exatamente como descrito na seção anterior, sendo a única diferença as matrizes de pesos utilizadas na projeção da matriz de entrada  $x$ , ou seja:

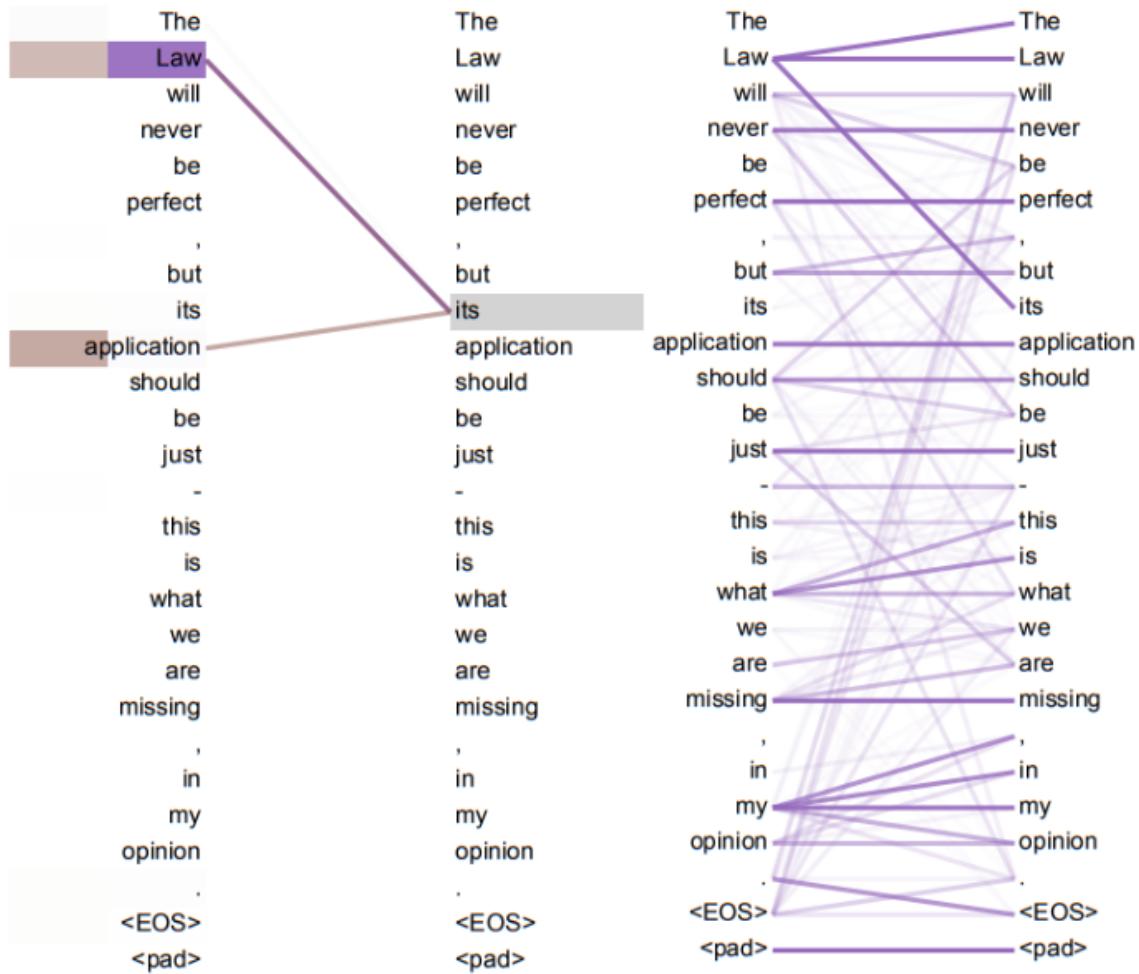
$$\text{head}_i = \text{Attention}(xW_i^Q, xW_i^K, xW_i^V) \quad (7.6)$$

A operação de concatenação das cabeças resulta em uma matriz pertencente ao  $\mathbb{R}^T \times \mathbb{R}^{hd}$ . Em seguida, uma projeção linear é realizada com a matriz de pesos  $W^O$ , que também é aprendida durante o *backpropagation*.



**Figura 7.2:** O diagrama de VASWANI et al., 2017 resume as operações da Atenção Paralela.

Intuitivamente, podemos interpretar cada cabeça como uma unidade de Atenção que foca em algum aspecto diferente da sequência de entrada. Por exemplo, podemos pensar que enquanto uma cabeça analisa a sintaxe da sentença, outra cabeça pode analisar as relações semânticas, enquanto uma terceira poderia observar figuras de linguagem.



**Figura 7.3:** À esquerda, exemplo retirado de VASWANI et al., 2017 mostra duas cabeças de Atenção (nota: cada linha colorida representa uma cabeça de Atenção) atendendo de maneira diferente à palavra "its". À direita, temos apenas uma cabeça de Atenção com a opacidade da cor representando maiores ou menores pesos de Atenção.

## 7.4 Aplicação no GPT

Abaixo mostramos a construção de uma classe em *Python* para uma única cabeça de Atenção. Note que para simplificar, optamos por não utilizar os viéses (*bias*) e os definimos como *False*. Além disso, definimos as matrizes de pesos  $W^Q$ ,  $W^K$  e  $W^V$  com dimensões associadas ao tamanho do vetor de *embedding* (*n\_embd*) e ao *head\_size*, que é simplesmente o resultado da divisão entre a dimensão do *embedding* e o total de cabeças de Atenção  $h$ . Também indicamos ao pacote *PyTorch* na linha 14 que a matriz que performará o mascaramento não precisa ser considerada durante o *backpropagation*. A linha 18 representa uma camada de *Dropout* que será explicada mais adiante.

```

1   class Head(nn.Module):
2       def __init__(self, head_size):
3           super().__init__()
4           # Inicializa as matrizes de chaves, consultas e valores
5           # Não inicializamos os viéses para simplificar o modelo
6           self.key = nn.Linear(n_embd, head_size, bias=False)
7           self.query = nn.Linear(n_embd, head_size, bias=False)
8           self.value = nn.Linear(n_embd, head_size, bias=False)
9           # Registra um buffer para a máscara de atenção triangular inferior
10          # Trata-se de matriz com 1's abaixo da diagonal principal (inclusive)
11          # e zeros acima. O uso de 'register_buffer' permite que o tensor 'tril',
12          # seja movido junto com o modelo para a GPU, se disponível, e
13          # não seja considerado um parâmetro do modelo.
14          self.register_buffer("tril", torch.tril(
15              torch.ones(context_len, context_len)
16          ))
17          # Inicializa a camada de Dropout
18          self.dropout = nn.Dropout(dropout)

```

Em seguida, definimos ainda dentro da classe, a função que realiza o passo *Forward* do treinamento. Note que as operações descritas na seção 7.2 estão devidamente codificadas na função, sendo que a única diferença está na inclusão de uma nova dimensão  $B$ , que representa o lote (*batch*) de sequências que estamos considerando. Mais detalhes sobre o *batch* serão fornecidos no capítulo sobre o treinamento.

```

1   # Define a passagem forward do modelo
2   def forward(self, x):
3       # Obtém as dimensões do conjunto de dados
4       batch_size, context_len, n_embd = x.shape
5       # Projeção linear da entrada através da matriz de pesos das chaves
6       k = self.key(x)
7       # Projeção linear da entrada através da matriz de pesos das consultas
8       q = self.query(x)
9       # Produto escalar entre q e k com dimensionamento  $\text{sqrt}(n_{\text{embd}})$ 
10      wei = q @ k.transpose(-2, -1) * n_embd ** (-0.5)
11      # Aplicação da máscara na matriz resultado da operação anterior
12      # Note que as entradas iguais a zero são redefinidas como  $-\inf$ 
13      wei = wei.masked_fill(
14          self.tril[:context_len, :context_len] == 0, float("-inf")
15      )
16      # Aplicação da função Softmax
17      wei = F.softmax(wei, dim=-1)

```

```

18     # Aplicação do Dropout
19     wei = self.dropout(wei)
20     # Projeção linear da entrada através da matriz de pesos dos valores
21     v = self.value(x)
22     # Multiplicação final pela matriz de valores
23     out = wei @ v
24     return out

```

Por fim, definimos outra classe, dessa vez para implementar a Atenção Paralela, que também é exatamente como descrita na seção anterior.

```

1  # Definição da classe de múltiplas cabeças
2  class MultiHeadAttention(nn.Module):
3      def __init__(self, num_head, head_size):
4          super().__init__()
5          # Inicializa uma lista de módulos
6          # com várias instâncias da cabeça de atenção
7          self.heads = nn.ModuleList([Head(head_size) for _ in range(num_head)])
8          # Inicializa uma camada linear de projeção dos dados
9          self.proj = nn.Linear(n_embd, n_embd)
10         # Inicializa a camada de Dropout
11         self.dropout = nn.Dropout(dropout)
12         # Define a passagem forward do modelo
13         def forward(self, x):
14             # Aplica as cabeças de atenção sobre os dados
15             # e as concatena no final
16             out = torch.cat([h(x) for h in self.heads], dim=-1)
17             # Realiza a projeção linear do resultado anterior
18             # e aplica o dropout em seguida
19             out = self.dropout(self.proj(out))
20             return out

```

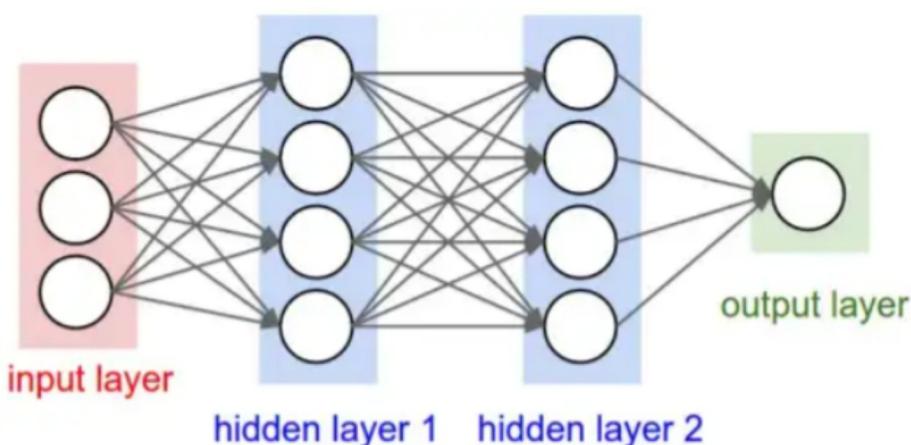
# Capítulo 8

## Rede Neural *Feed-Forward*

Após o processamento da sequência de entrada pelo mecanismo de Atenção, o GPT emprega uma rede neural do tipo *Feed-Forward* (FFNN, do inglês *Feed-Forward Neural Network*) para continuar transformando a informação.

A combinação de uma camada de Atenção com uma rede FFNN em seguida é denominada um bloco do *Transformer*. No GPT, os blocos são empilhados sequencialmente e a escolha do número de blocos que compõe o modelo é um hiperparâmetro. Na Figura 1.1, este hiperparâmetro é denotado por  $N_x$ , número que se refere à profundidade do número de camadas (*layers*) do modelo. No maior modelo do GPT-3, por exemplo, existem 96 blocos.

O papel das redes FFNN no GPT é projetar os dados para um espaço dimensional superior, permitindo ao modelo criar interações complexas, antes de finalmente projetar os dados de volta para a mesma dimensão do espaço das entradas pertencente ao  $\mathbb{R}^d$ . Esta operação é executada independente e paralelamente para cada posição da sequência.



**Figura 8.1:** No diagrama, retirado de [MUKUL RATHI, 2018](#), temos a arquitetura de uma FFNN simples. No GPT, essa estrutura é muito maior, potencialmente com milhares de nós intermediários.

## 8.1 Estrutura da FFNN

No contexto do GPT, a arquitetura da rede neural *Feed-Forward* consiste de duas camadas lineares (também conhecidas como camadas ocultas (*hidden layers*)) com uma função de ativação não-linear entre elas.

As camadas lineares são operações que envolvem matrizes de pesos, as quais são aprendidas durante o *backpropagation*. A primeira camada linear expande as dimensões dos dados de entrada, enquanto que a segunda as comprime de volta. A expansão permite que a rede crie representações mais complexas e aumenta a capacidade do modelo de aprender relações profundas.

Matematicamente, estas operações são denotadas por:

$$F_1(\mathbf{x}) = \text{ReLU}(\mathbf{x}W_1 + b_1) \quad (8.1)$$

$$F_2(\mathbf{x}) = F_1(\mathbf{x})W_2 + b_2 \quad (8.2)$$

onde:

- $\mathbf{x} \in \mathbb{R}^T \times \mathbb{R}^d$  são as representações de entrada que vieram da camada de Atenção.
- $W_1 \in \mathbb{R}^d \times \mathbb{R}^{d_{ff}}$  e  $b_1 \in \mathbb{R}^{d_{ff}}$  são os pesos e os vieses da primeira camada linear.
- $W_2 \in \mathbb{R}^{d_{ff}} \times \mathbb{R}^d$  e  $b_2 \in \mathbb{R}^d$  são os pesos e os vieses da segunda camada linear.
- $\text{ReLU}$  é a função não linear de ativação (do inglês, *Rectified Linear Unit*), definida por  $\text{ReLU}(\mathbf{x}) = \max(0, \mathbf{x})$ .
- $d_{ff}$  é a dimensão intermediária da FFNN (tipicamente, no GPT,  $d_{ff} = 4d$ ).

A função de ativação *ReLU* transforma qualquer entrada negativa em zero e mantém inalterada qualquer entrada positiva. Ela é escolhida por sua eficiência computacional e propriedades que ajudam na convergência durante o treinamento.

## 8.2 Aplicação no GPT

A implementação da FFNN no GPT do Apêndice A é bastante simples graças à biblioteca *PyTorch* que automaticamente lida com a construção das matrizes e dos viéses.

```

1 # Define a classe da rede Feed Forward
2 class FeedForward(nn.Module):
3     def __init__(self, n_embd):
4         super().__init__()
5         # Cria a arquitetura da rede com uma camada linear,
6         # seguida de uma ativação ReLU, seguida por outra
7         # camada linear e, por fim, com uma camada de dropout
8         self.net = nn.Sequential(
9             nn.Linear(n_embd, 4 * n_embd),
10            nn.ReLU(),
11            nn.Linear(4 * n_embd, n_embd),

```

```
12         nn.Dropout(dropout),  
13     )  
14     # Define a passagem forward do modelo  
15     def forward(self, x):  
16         # Aplica a sequência de camadas definida para a FFNN  
17         return self.net(x)
```

# Capítulo 9

## Componentes Auxiliares

Até agora, podemos dizer que exploramos as principais componentes que formam o GPT. Contudo, outras camadas e operações - sendo algumas destas propostas no artigo original do Transformer e outras acrescentadas posteriormente por pesquisadores da área - também possuem papel essencial na arquitetura.

Neste capítulo, iremos descrever três componentes que ajudam a promover estabilidade e eficiência ao modelo, além de otimizar o treinamento e aumentar a performance de generalização. São elas: a camada de normalização, as conexões residuais e a camada de *Dropout*.

### 9.1 Camada de Normalização

A arquitetura do GPT faz uso extensivo da técnica de Normalização de Camadas (*Layer Normalization*), proposta por [BA et al., 2016](#) e inspirada em outra técnica chamada de Normalização de Lotes (*Batch Normalization*) [IOFFE e SZEGEDY, 2015](#).

A *Layer Norm*, como é comumente denominada, é fundamental para estabilizar as ativações (leia-se, vetores que carregam as informações) ao longo da rede. Modelos de *deep learning*, como o GPT, são propensos a apresentar grandes oscilações numéricas devido à inicialização randômica dos pesos e cálculos do gradiente. Esses eventos podem comprometer significativamente o treinamento do modelo e, consequentemente, o desempenho final do modelo.

A operação de Normalização utiliza a média  $\mu$  e o desvio padrão  $\sigma$  das componentes que formam a representação vetorial de cada *token* para normalizar cada elemento do vetor, conforme a equação:

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \quad (9.1)$$

onde:

- $\hat{x}_{i,j}$  - O valor normalizado do  $j$ -ésimo elemento do vetor que representa o  $i$ -ésimo *token*.

- $x_{i,j}$  - Representa o  $j$ -ésimo elemento do vetor do  $i$ -ésimo *token*.
- $\mu_i$  - A média de todos os elementos do vetor do  $i$ -ésimo *token*.
- $\sigma_i$  - O desvio padrão de todos os elementos do vetor do  $i$ -ésimo *token*.
- $\epsilon$  - Uma constante pequena para garantir estabilidade numérica e evitar divisão por zero.

Esta operação resulta em uma distribuição com média 0 e variância 1. No entanto, para permitir que o modelo aprenda a distribuição ideal dos dados, são introduzidos dois parâmetros treináveis durante o *backpropagation*,  $\gamma$  e  $\beta$ . Estes parâmetros ajustam a normalização para que o modelo possa efetivamente aprender a média e a variância ótimas para as ativações. Matematicamente, esta etapa é representada por:

$$y_i = \gamma \hat{x}_i + \beta, \quad (9.2)$$

onde  $\gamma$  e  $\beta$  são parâmetros que representam, respectivamente, o fator de escala (*scale*) e deslocamento (*shift*) aplicados à distribuição normalizada.

No GPT, a normalização é aplicada em três momentos:

1. Antes da camada de Atenção;
2. Antes da rede neural *Feed-Forward*;
3. Antes da camada linear final do modelo.

Vale notar que a implementação da *Layer Norm* após a soma dos *embeddings* com as codificações posicionais, ou seja, antes da camada de Atenção, não está presente no artigo de [VASWANI \*et al.\*, 2017](#) e foi introduzida mais recentemente por pesquisadores da área que notaram empiricamente a melhora no desempenho do modelo após esta modificação na arquitetura original.

## 9.2 Conexões Residuais

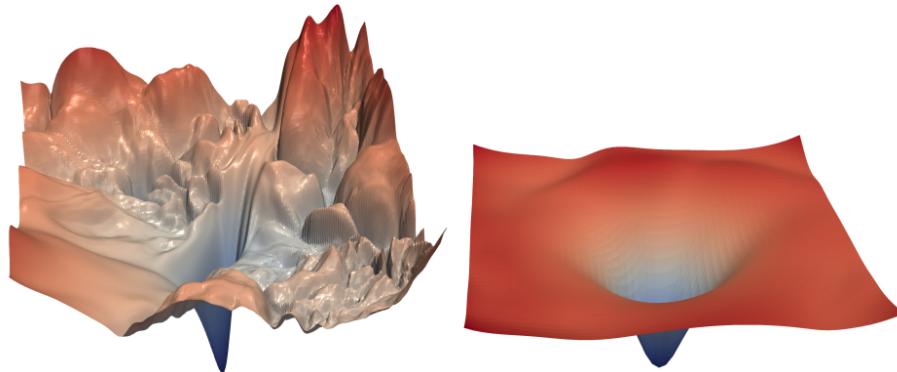
As Conexões Residuais (*Skip Connections*) compreendem uma operação bastante simples, porém muito importante nas arquiteturas de aprendizado de máquina profundo. Introduzidas por [He \*et al.\*, 2015](#), a implementação desta estratégia, não apenas em modelos de linguagem, como também em modelos de visão computacional, se tornou onipresente.

A ideia básica das Conexões Residuais consiste em somar a entrada original de alguma camada com a saída daquela entrada após ser processada. Com isso, cria-se um caminho alternativo para o fluxo dos gradientes que serão calculados durante o treinamento. Isso facilita o treinamento e reduz o efeito do desaparecimento dos gradientes.

A operação é matematicamente simples e direta, envolvendo somente a adição da entrada original à saída processada:

$$y = f(x) + x, \quad (9.3)$$

onde  $x$  é a entrada e  $f(x)$  representa a transformação realizada pela camada.



**Figura 9.1:** No diagrama, retirado de [Hao Li, 2018](#), à esquerda temos uma superfície da função de perda de um modelo de imagem sem a presença de conexões residuais. À direita, a superfície da função de perda da mesma rede com a adição das conexões, muito mais suave e fácil de ser otimizada.

No GPT, as conexões residuais são aplicadas em dois momentos:

1. Entre a saída da camada de Atenção e a sua entrada ainda normalizada;
2. Entre a saída da FFNN e a sua entrada não normalizada.

### 9.3 Dropout

O *Dropout* ou camada de *Dropout* é uma técnica de regularização introduzida por [Srivastava et al., 2014](#) que se tornou uma estratégia fundamental para evitar o sobreajuste (*overfitting*) em redes neurais, como o GPT. Essa técnica consiste em desativar aleatoriamente um subconjunto das ativações durante a fase de treinamento. Com isso, espera-se que o modelo não dependa excessivamente de qualquer neurônio individualmente e, consequentemente, aumente a performance de generalização.

Matematicamente, o *Dropout* é aplicado a um vetor de entrada  $x$  com uma probabilidade  $p$  de cada elemento deste vetor ser zerado. Seja  $G$  uma matriz cujas entradas são variáveis aleatórias Bernoulli com parâmetro  $p$  de dimensões iguais a  $x$ . Ou seja, cada entrada vale 1 com probabilidade  $1 - p$  e 0 com probabilidade  $p$ . A saída  $y$  da operação de *Dropout* é dada por:

$$y = G \odot x, \quad (9.4)$$

onde  $\odot$  denota a multiplicação elemento a elemento.

Durante a fase de treinamento, para compensar o efeito do *Dropout*, as ativações são multiplicadas pelo fator  $1 - p$ , garantindo que a esperança da saída seja a mesma tanto durante o treinamento quanto durante a inferência.

O *Dropout* é aplicado nas seguintes etapas do GPT:

- Na matriz com os pesos de Atenção normalizados pela *Softmax*, ou seja, antes de multiplicarmos pela matriz de Valores  $V$ .
- Depois da projeção linear da concatenação originada das múltiplas cabeças de Atenção.
- Ao final da FFNN, ou seja, depois que os dados são reprojetados para a dimensão original dos *embeddings*.

# Capítulo 10

## Gerando Probabilidades

Após executar todas as transformações de dados por meio dos *embeddings*, codificações posicionais, camada de normalização, Mecanismo de Atenção, conexões residuais e a rede *Feed-Forward*, o modelo GPT está a apenas um passo do ponto crucial de todo o processo: a geração de probabilidades para o próximo *token* da sequência.

Esta etapa derradeira é a projeção dos dados por meio de uma camada Linear, que irá produzir escores, ou logits no linguajar técnico, que serão finalmente transformados em probabilidades após a aplicação da função *Softmax*. Vejamos a seguir mais detalhes sobre essas operações.

### 10.1 Camada Linear

A camada Linear final do GPT é responsável por transformar as representações de *tokens* que foram processadas e refinadas pelas camadas anteriores em escores brutos, que podem ser interpretados como logits.

Matematicamente, a camada linear é uma transformação afim dada por:

$$z = \mathbf{x}W^f + b, \quad (10.1)$$

onde  $\mathbf{x} \in \mathbb{R}^T \times \mathbb{R}^d$  são as representações dos *tokens* que refletem todas as informações das palavras após passarem por todo o pipeline do GPT,  $W^f \in \mathbb{R}^d \times \mathbb{R}^{N_v}$  é a matriz de pesos da camada linear final,  $b \in \mathbb{R}^{N_v}$  é o vetor de bias, e  $z \in \mathbb{R}^T \times \mathbb{R}^{N_v}$  são os escores de logits.

Cada linha  $i$  da matriz  $z$  representa as pontuações para todos os possíveis próximos *tokens* em relação ao *token* na posição  $i$  da sequência de tamanho  $T$ . Observe que a dimensão das colunas da saída desta camada é igual ao tamanho do vocabulário  $N_v$ , pois estaremos atribuindo um escore para cada possível próximo *token* do vocabulário.

## 10.2 Softmax

Durante a fase de inferência, quando queremos gerar texto a partir de um *prompt* de entrada, aplicamos a função *Softmax*, que converte a última linha da matriz  $z$  de logits, aquela que considera todo o *prompt* de entrada, em uma distribuição de probabilidade válida, fornecendo as probabilidades para todos os *tokens* do vocabulário.

No contexto a que nos referimos, a *Softmax* é aplicada da seguinte maneira:

$$\text{Softmax}(z(T, :)) = \frac{e^{z_i}}{\sum_{j=1}^{N_v} e^{z_j}}, \quad (10.2)$$

para  $i = 1, \dots, N_v$ .

A função exponencial no numerador garante que todos os valores de saída sejam positivos, e a normalização no denominador assegura que a soma de todas as probabilidades seja igual a 1, característica essencial de uma distribuição de probabilidade.

A saída da *Softmax*, um vetor da mesma dimensão do vocabulário, fornece a probabilidade  $P(x_{T+1}|x_T, \dots, x_1)$ , assumindo que  $x_{T+1}$  é o próximo token e a sequência atual é  $(x_1, \dots, x_T)$ .

É crucial notar que a *Softmax* é uma função que amplifica as diferenças entre os valores de entrada; por exemplo, pequenas diferenças entre escores de logits são exponencialmente ampliadas. Isso significa que mesmo se alguns *tokens* tiverem escores brutos similares, a função fará com que as discrepâncias em probabilidades sejam significativas.

Abaixo, podemos ver como está implementada a função *Softmax* no GPT do Apêndice A. Observe ainda que utilizamos a distribuição Multinomial para amostrar o próximo *token* mais provável ao invés de simplesmente tomarmos o *token* com maior probabilidade.

```

1  # Gera novas sequências de texto
2  def generate(self, idx, max_new_tokens):
3      # Itera até o tamanho máximo de tokens a ser gerado
4      for _ in range(max_new_tokens):
5          # Limita a entrada ao tamanho da janela de contexto
6          idx_cond = idx[:, -context_len:]
7          # Calcula os logits para o contexto de entrada
8          logits, loss = self(idx_cond)
9          # Extrai os logits
10         logits = logits[:, -1, :]
11         # Calcula as probabilidades com a Softmax
12         probs = F.softmax(logits, dim=-1)
13         # Sorteia o próximo token a partir de uma distribuição
14         # multinomial. Aqui, obtemos apenas uma amostra (token)
15         idx_net = torch.multinomial(probs, num_samples=1)
16         # Concatena o novo token ao contexto de entrada
17         idx = torch.cat((idx, idx_net), dim=1)
18     return idx

```

# Capítulo 11

## Pré-treinamento

Agora que já exploramos cada uma das componentes do GPT, iremos analisar como utilizar a arquitetura para realizar o pré-treinamento do modelo.

De maneira intuitiva, esta etapa resume-se a apresentar ao "robô" uma grande quantidade de dados de treinamento que serão utilizados para a extração dos padrões da linguagem e, consequentemente, irão possibilitar a realização da tarefa de previsão do próximo *token* na sequência.

Por se tratar de dados não rotulados explicitamente, pois os rótulos (*labels*) são obtidos diretamente dos textos, esta abordagem é chamada de aprendizado auto-supervisionado (*self-supervised learning*). Como o objetivo é prever a próxima palavra, podemos simplesmente fornecer ao modelo uma sentença com a palavra final faltante para que ele aprenda e forneça uma previsão.

Idealmente, a base de dados utilizada nesta fase do treinamento deve ser vasta e diversa para promover o aprendizado de uma ampla gama de estruturas textuais e contextos linguísticos.

É durante o pré-treinamento que o modelo aprende a formar palavras e sentenças que façam sentido gramatical e semanticamente. Contudo, é importante frisar que esta habilidade apenas começa a emergir se estivermos considerando um conjunto de dados relativamente grande e uma arquitetura do modelo também grande. Vale a pena revisitar a tabela 2.9 que mostra o tamanho da arquitetura para o GPT-3. Especula-se que as versões posteriores dos modelos da OpenAI sejam ainda muito maiores.

No código abaixo retirado do GPT apresentado no Apêndice A, realizamos a indexação de todos os dados, conforme explicado na seção 4.2, e separamos 90% como conjunto de treinamento e o restante como conjunto de validação. Para simplificar, não utilizamos um conjunto de teste.

```

1 # Codificação em índices dos dados de treinamento
2 data = torch.tensor(encode(text), dtype=torch.long)
3 # Divisão dos dados em treino e validação
4 train_percent = 0.9
5 n = int(train_percent * len(data))
6 train_data = data[:n]
7 val_data = data[n:]

```

Em seguida, definimos uma função que irá selecionar aleatoriamente uma determinada quantidade de lotes (*batch\_size*) de sequências do conjunto de dados. A quantidade de lotes também é um hiperparâmetro do modelo que deve ser definido previamente.

Depois, para cada sequência, utilizando a técnica de janela deslizante descrita na seção 4.3, criamos os exemplos de treinamento  $x$  e os alvos (*targets*), denominados por  $y$  no código abaixo. Note que todos os exemplos possuem tamanho  $T$ , que corresponde à janela de contexto.

A razão para separar os dados em lotes de treinamento está relacionada puramente à eficiência computacional.

```

1 # Função para obter lotes de dados
2 def get_batch(split):
3     # Seleciona se os dados virão do conjunto de treino ou validação
4     data = train_data if split == "train" else val_data
5     # Gera batch_size sequências aleatórias de índices iniciais para cada
6     # lote
7     ix = torch.randint(len(data) - context_len, (batch_size,))
8     # Gera sequências de treinamento baseadas no tamanho da janela de
9     # contexto
10    x = torch.stack([data[i : i + context_len] for i in ix])
11    # Gera rótulos para as sequências de treinamento
12    y = torch.stack([data[i + 1 : i + context_len + 1] for i in ix])
13    # Envia os tensores para a GPU ou CPU
14    x, y = x.to(device), y.to(device)
15    return x, y

```

## 11.1 Função de Perda

O objetivo central do pré-treinamento é ajustar os parâmetros  $\Theta$ , ou seja, os pesos das matrizes da arquitetura Transformer, com o objetivo de minimizar uma função de perda. No âmbito do GPT e de vários outros modelos de aprendizado de máquina (machine learning), utiliza-se a entropia cruzada (cross-entropy) como a função de perda  $\mathcal{L}$  associada ao problema de estimativa da distribuição. Abaixo mostramos a fórmula considerando apenas uma sequência (um lote):

$$\mathcal{L}(\Theta) = - \sum_{t=1}^T \ln \hat{P}_{\Theta}(y_t | x_1, x_2, \dots, x_{t-1}) \quad (11.1)$$

Nota: Se tivéssemos mais de uma sequência, tomariímos a média entre os lotes como resultado da perda.

Ao minimizarmos a equação 11.1 com relação aos parâmetros, obtemos como subproduto a estimativa  $\hat{P}(\mathbf{x})$  que buscávamos.

A entropia cruzada, também conhecida como *log loss*, é uma métrica usada para avaliar o desempenho de modelos de classificação cuja saída é um valor de probabilidade entre 0 e 1. Ela aumenta à medida que a probabilidade prevista se desvia do rótulo real.

A função abaixo é utilizada para calcular a estimativa média da função de perda durante o processo de treinamento.

```

1 # Decorador para evitar o cálculo dos gradientes durante o cálculo da perda
2 @torch.no_grad()
3 # Função que calcula a perda
4 def estimate_loss():
5     out = []
6     # Coloca o modelo em modo de avaliação
7     model.eval()
8     # Itera sobre os conjuntos de dados
9     for split in ["train", "val"]:
10         # Inicializa o vetor de perdas com zeros
11         losses = torch.zeros(eval_iters)
12         # Itera sobre o número de avaliações pré-determinadas
13         for k in range(eval_iters):
14             # Obtém exemplos aleatórios de sequências dos conjuntos de dados
15             X, Y = get_batch(split)
16             # Obtém os logits e a perda para os exemplos selecionados
17             logits, loss = model(X, Y)
18             # Atualiza o vetor de perdas
19             losses[k] = loss.item()
20         # Calcula a média da perda para cada conjunto de dados
21         out[split] = losses.mean()
22     # Recoloca o modelo em modo de treinamento
23     model.train()
24     return out

```

## 11.2 Otimização dos Parâmetros

A otimização dos parâmetros  $\Theta$  em modelos de aprendizado profundo de máquina, como o GPT, é um processo fundamental que pode ser realizado por meio de diferentes algoritmos. O mais básico é a Descida do Gradiente Estocástico (SGD, do inglês *Stochastic Gradient Descent*), e uma de suas variações avançadas é o otimizador AdamW. Vejamos abaixo os detalhes dessas abordagens:

**Stochastic Gradient Descent:** O SGD é um método iterativo para otimizar uma função de perda. A ideia básica por trás do método é atualizar os parâmetros de forma iterativa na direção oposta ao gradiente da função de perda. A fórmula de atualização é:

$$\Theta_{\text{prox}} \leftarrow \Theta - \eta \nabla_{\Theta} \mathcal{L}(\Theta), \quad (11.2)$$

onde  $\Theta_{\text{prox}}$  são os parâmetros atualizados,  $\Theta$  são os parâmetros atuais,  $\eta$  é a taxa de aprendizado, e  $\nabla_{\Theta} \mathcal{L}(\Theta)$  é o gradiente da função de perda com relação aos parâmetros.

O algoritmo é considerado estocástico porque ele atualiza os pesos do modelo com base apenas nos exemplos de treinamento constantes no *batch* que está sendo processado e não em todo o conjunto de dados, como seria o caso na clássica Descida do Gradiente (*Gradient Descent*).

**AdamW:** O AdamW é uma extensão do otimizador Adam (*Adaptive Moment Estimation*), que ajusta a taxa de aprendizado de cada parâmetro com base nos momentos de primeira e segunda ordem dos gradientes. A fórmula de atualização para AdamW é mais complexa do que a do SGD e pode ser resumida da seguinte forma:

$$\Theta_{\text{prox}} \leftarrow \Theta - \left( \frac{\eta}{\sqrt{\hat{v}} + \epsilon} \hat{m} + \lambda \Theta \right), \quad (11.3)$$

onde  $\hat{m}$  e  $\hat{v}$  são, respectivamente, estimativas corrigidas dos primeiros e segundos momentos dos gradientes,  $\lambda$  é a taxa de decaimento dos pesos e  $\epsilon$  é um número pequeno para evitar divisão por zero. AdamW modifica o Adam adicionando a regularização de peso diretamente no termo de atualização dos parâmetros, oferecendo uma abordagem mais eficaz para a regularização.

No GPT do Apêndice A, utilizamos o AdamW como otimizador.

**Backpropagation:** Fundamental para o treinamento de redes neurais, o algoritmo de retropropagação (*backpropagation*) é o processo de calcular o gradiente da função de perda em relação a cada parâmetro utilizando a regra da cadeia. Isso é realizado através da propagação reversa do erro, começando da saída e movendo-se para trás através das camadas da rede. A fórmula básica do *backpropagation* para uma única camada é:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial \Theta}, \quad (11.4)$$

onde  $\frac{\partial \mathcal{L}}{\partial a}$  é o gradiente da função de perda com relação à ativação da camada, e  $\frac{\partial a}{\partial \Theta}$  é o gradiente da ativação com relação aos parâmetros da camada.

Abaixo, podemos ver o loop principal do pré-treinamento do modelo GPT apresentado no Apêndice A. Observe que devemos escolher uma taxa de aprendizado (*learning rate*) para o nosso otimizador. Na penúltima linha do código, executamos o algoritmo de retropropagação.

```

1 # Inicializa o modelo
2 model = GPTLanguageModel()
3 # Move o modelo para o dispositivo apropriado (GPU/CPU)
4 m = model.to(device)
5 # Configura o otimizador com taxa de aprendizagem específica
6 optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
7
8 def train():
9     start = time.time()
10
11     # Barra de progresso do treinamento
12     progress_bar = tqdm(total=max_iters, desc="Progresso do treinamento")
13
14     # Loop de treinamento para max_iters iterações
15     for iter in range(max_iters):
16         # Verifica se é hora de estimar a perda e exibir informações
17         if iter % eval_interval == 0 or iter == max_iters-1:
18             # Calcula e exibe as perdas
19             losses = estimate_loss()
```

```

20
21     print(f"\nIteração {iter}:",
22           f"Perda de Treino {losses['train']:.4f}",
23           f"Perda de Validação {losses['val']:.4f}")
24
25     # Gera novo texto a partir do modelo
26     context = torch.zeros((1, 1), dtype=torch.long, device=device)
27     output_gerado = decode(
28         model.generate(context, max_new_tokens=512)[0].tolist()
29     )
30     print(f"\nTexto gerado pelo modelo na iteração: {iter}")
31     print(output_gerado)
32
33     # Salva checkpoint do modelo
34     caminho_salvamento_modelo = f"modelo_{iter}.pth"
35     torch.save(model.state_dict(), caminho_salvamento_modelo)
36
37     # Obtém lotes de dados de treinamento
38     xb, yb = get_batch("train")
39     # Realiza o passo à frente do modelo, calculando os logits e loss
40     logits, loss = model(xb, yb)
41     # Zera os gradientes dos parâmetros do modelo
42     optimizer.zero_grad(set_to_none=True)
43     # Calcula os gradientes da perda em relação aos parâmetros
44     # Backpropagation
45     loss.backward()
46     # Atualiza os parâmetros do modelo usando o otimizador
47     optimizer.step()
48
49     # Atualiza barra de progresso
50     progress_bar.update(1)
51
52     # Encerra barra de progresso
53     progress_bar.close()
54
55     end = time.time()
56     total_time = end - start
57     print(f"\nTempo total de treinamento em segundos: {float(total_time):.2f}
58     ")
59     print("\n")

```

Uma vez pré-treinado, o GPT ainda não possui todas as capacidades a que estamos acostumados quando utilizamos, por exemplo, o ChatGPT da OpenAI. Para isso, ainda precisamos "alinhar" o modelo para que ele responda de maneira satisfatória às preferências dos seres humanos. Abordaremos essa etapa no próximo capítulo. Entretanto, após o pré-treinamento, o modelo é capaz de realizar o que a literatura denominada como aprendizado por contexto (*in-context learning*). Podemos, por exemplo, dar exemplos ao modelo e solicitar que ele realize a tarefa conforme mostrado.

Por exemplo, quando estivermos gerando tokens do modelo, podemos escrever o seguinte *prompt* de comando:

Maçã -> Apple

Abacaxi ->

O modelo, então, entenderá "sozinho" que deverá completar com a palavra correspondente à Abacaxi na língua inglesa: "*Pineapple*".

Este método, contudo, pode ser considerado menos sofisticado quando comparado a um modelo que passou por técnicas adicionais de alinhamento, as quais serão discutidas adiante.

# Capítulo 12

## Alinhamento

Treinar modelos de linguagem maiores não os tornam inherentemente melhores em seguir a intenção de um usuário. Por exemplo, grandes modelos de linguagem podem gerar respostas que são inverídicas, tóxicas ou simplesmente não úteis para o usuário. Em outras palavras, esses modelos não estão alinhados com as intenções dos seus usuários. [Ouyang et al., 2022](#)

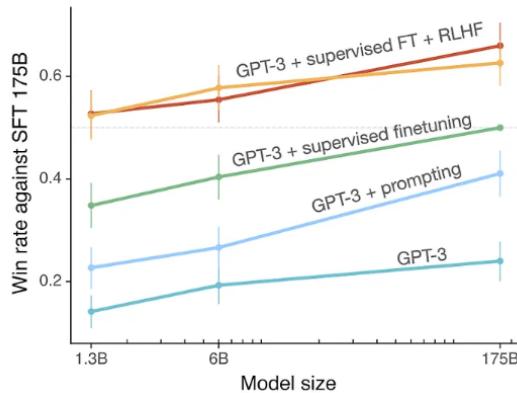
Após a fase crucial de pré-treinamento, um serviço como o ChatGPT precisa passar por mais algumas etapas antes de ser colocado diante dos usuários. Neste capítulo, iremos descrever de forma geral e breve quais são os métodos de "alinhamento" utilizados para transformar um modelo pré-treinado, que é basicamente um regurgitador de conteúdo, em um modelo que pode ser utilizado para resolver tarefas realmente úteis aos seres humanos.

Em linhas gerais, essas etapas incluem o refinamento por instrução (*instruction fine-tuning*), a criação de um modelo de recompensa baseado em preferências dos humanos (*reward model*) e, por fim, um novo refinamento, processo conhecido como aprendizado por reforço com *feedback humano* (RLHF, do inglês *Reinforcement Learning with Human Feedback*).

### 12.1 Refinamento por Instrução

O refinamento por instrução consiste, basicamente, em continuar o treinamento do modelo pré-treinado, agora de maneira supervisionada, com base em um conjunto de dados composto por pares de perguntas (ou instruções) e respostas que foram, idealmente, construídos por seres humanos. Esse conjunto de dados é, geralmente, muito menor que a massa textual utilizada durante o pré-treinamento e possui dezenas de milhares de exemplos apenas.

Nessa fase, são fornecidos ao modelo exemplos de tarefas juntamente com as instruções explícitas, incentivando-o a seguir diretivas de maneira mais eficaz. Essa técnica se diferencia do pré-treinamento por focar em obedecer comandos de forma útil e segura, abordando desafios como a geração de textos tendenciosos ou inverídicos. O refinamento por instrução melhora a precisão das respostas e também a relevância destas.



**Figura 12.1:** No diagrama, retirado de [OUYANG et al., 2022](#) vemos a comparação da eficácia de diferentes abordagens de alinhamento em modelos GPT-3 de vários tamanhos. A primeira linha debaixo para cima mostra o GPT-3 pré-treinado, em seguida o mesmo modelo com a técnica de prompting (*in-context learning*). A terceira linha mostra a performance do GPT-3 após o refinamento por instrução. Por último, as duas linhas de cima do gráfico mostram a combinação desta última técnica com o RLHF em diferentes cenários envolvendo o algoritmo de otimização.

## 12.2 Modelo de Recompensa

Uma vez realizado o refinamento, gera-se uma quantidade de amostras de respostas para um mesmo *prompt* oriundas desse modelo com o objetivo de apresentá-las a seres humanos que possam ranqueá-las em ordem de preferência. Com base nisso, então, treina-se um modelo de recompensa, ou seja, um sistema de pontuação que reflete quais respostas são mais desejáveis ou alinhadas com os critérios estabelecidos, como veracidade, imparcialidade e não-toxicidade.

Este modelo de recompensa serve como uma função objetivo para o processo de aprendizado por reforço subsequente, orientando o modelo de linguagem a produzir respostas que são não apenas linguisticamente corretas, mas também alinhadas com valores humanos essenciais.

## 12.3 Aprendizado por Reforço com Feedback Humano

O RLHF é a etapa final de ajuste fino do modelo de linguagem para melhor alinhamento com as intenções do usuário. Neste processo, o modelo é exposto a uma série de tarefas onde ele precisa gerar respostas que são posteriormente avaliadas com base no modelo de recompensa previamente estabelecido.

As respostas alinhadas com as preferências humanas são recompensadas, enquanto as outras são penalizadas. Este ciclo de *feedback* permite que o modelo refine ainda mais sua capacidade de seguir instruções e produzir respostas que são corretas, coerentes, éticas, imparciais e úteis. Com o RLHF, o modelo aprende a navegar com maior precisão pelas nuances das interações humanas, tornando-se mais confiável e valioso aos usuários.

# Capítulo 13

## Conclusão

Com o objetivo de explicar o funcionamento de um modelo de linguagem como o ChatGPT, este trabalho ofereceu uma visão dos componentes matemáticos e computacionais que constituem o *decoder* do *Transformer*, que representa o modelo conhecido como *Generative Pre-trained Transformer* (GPT, na sigla em inglês), que ganhou bastante fama depois do advento do chatbot lançado pela empresa americana OpenAI.

A partir da introdução histórica, que contextualizou o surgimento e a evolução das técnicas de modelagem de linguagem até o advento do Transformer, estabelecemos uma base para compreender as inovações introduzidas pelo GPT. Ao longo dos capítulos, detalhamos cada uma das peças que compõem o modelo. Descrevemos as operações essenciais que permitem ao GPT processar e gerar linguagem natural: a "tokenização", que divide o texto em unidades manipuláveis; os *embeddings*, que incorporam os significados semânticos dos *tokens*; a codificação posicional, que garante a compreensão da sequencialidade no texto; além dos algoritmos de Atenção, de redes neurais *Feed-Forward* e dos mecanismos de normalização e conexões residuais, que juntos possibilitam que o GPT capture relações contextuais profundas.

A fase de pré-treinamento foi destacada como a etapa em que o modelo adquire conhecimento geral sobre a língua, sendo refinado e realinhado posteriormente via aprendizado supervisionado e RLHF, para garantir que suas respostas se alinhem às expectativas humanas.

Além disso, neste trabalho, implementamos e treinamos um GPT com 28,5 milhões de parâmetros com base na obra do escritor Machado de Assis. Com apenas uma GPU NVIDIA A100 (40GB) e uma base de dados relativamente pequena para os padrões do aprendizado de máquina profundo, mostramos que é possível obter resultados interessantes. Possivelmente, aos leitores atentos de Machado de Assis, tais excertos gerados a partir do modelo soem como "sonhos Machadianos".

Como futuros desdobramentos deste trabalho, poderíamos realizar procedimentos de *data augmentation*, ou seja, criação sintética de mais exemplos a partir da obra de Machado de Assis, com o objetivo de aumentar a base de dados. Além disso, se disponíveis os devidos recursos computacionais, poderíamos experimentar com modelos maiores, com mais parâmetros.

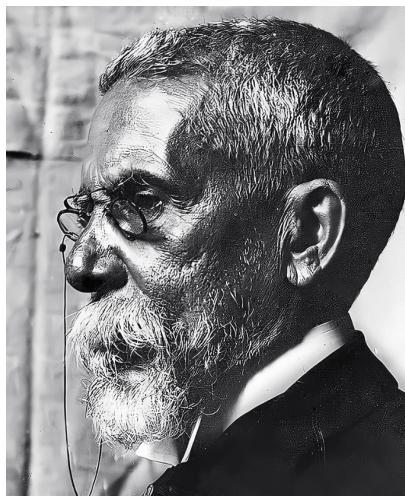
A pergunta de pesquisa que fica é: dado um conjunto de dados pequeno, porém muito rico (escrito por aquele que talvez seja considerado o maior escritor brasileiro de todos os tempos), e dada a disponibilidade de recursos computacionais, qual o ganho de performance que conseguiríamos obter?

Em suma, a jornada através do GPT revela não apenas um avanço tecnológico impressionante, mas também um convite à reflexão sobre o futuro que desejamos construir com essas poderosas ferramentas que desafiam os limites da Inteligência Artificial.

## Apêndice A

### Implementação em Python

Neste trabalho, utilizamos a obra completa do escritor brasileiro Machado de Assis como base de dados para a construção passo a passo de um modelo do tipo GPT.



**Figura A.1:** *Machado de Assis (1839-1908) é considerado um dos maiores escritores em língua portuguesa. (Foto: Arquivo Nacional. Autor desconhecido)*

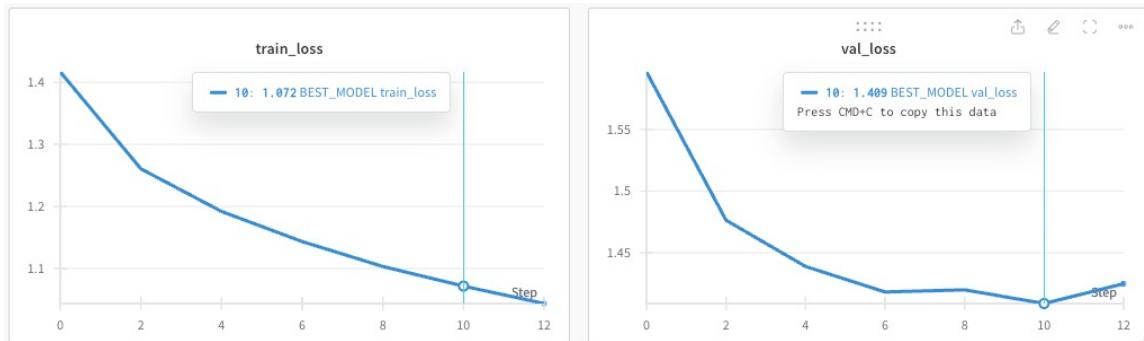
Utilizamos um dataset do *Kaggle* (<https://www.kaggle.com/datasets/luxedo/machado-de-assis>) que agrupa as 116 obras de ficção e outros textos de Machado de Assis nas categorias: contos, crítica, crônica, miscelânea, poesias, romances, teatro e traduções. As obras são de domínio público e estão publicadas em <http://machado.mec.gov.br>.

Para começar, concatenamos todos os textos em um único arquivo `textos_concatenados.txt`, que é considerado nosso *corpus* textual, de onde derivamos nosso vocabulário  $\mathcal{V}$ . Para simplificar, decidimos "tokenizar" o conjunto com base nos caracteres únicos. Ou seja, nosso modelo, prevê o próximo caractere com base nos últimos precedentes.

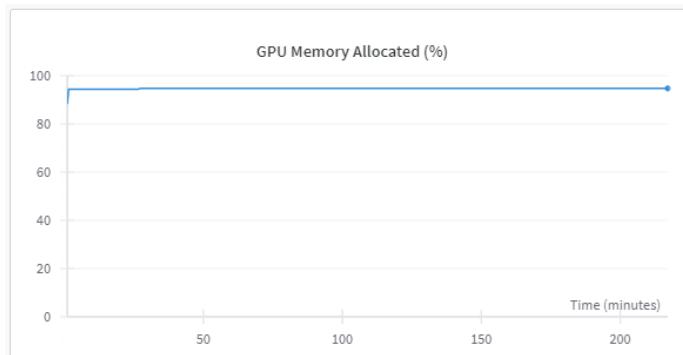
Abaixo, temos algumas estatísticas descritivas do corpus:

- Número de caracteres no arquivo: 10.977.697
- Número de caracteres únicos (excluindo espaços em branco e quebras de linha): 143
- Número de palavras únicas: 74.486

Para o treinamento do modelo, utilizamos uma única GPU (NVIDIA A100) com 40GB de memória. Além disso, para a escolha dos hiperparâmetros, realizamos busca aleatória no espaço de hiperparâmetros, que consumiu cerca de 100 horas de computação. O pré-treinamento do modelo, uma vez escolhidos os hiperparâmetros, durou cerca de 8 horas. O total de parâmetros do modelo é de 28,5 milhões.



**Figura A.2:** Resultados da função de perda nos conjuntos de treino e validação.



**Figura A.3:** Nossos modelos conseguiram aproveitar quase toda a memória disponível da GPU que possuímos para o pré-treinamento.

Durante o treinamento, geramos algumas sequências de texto a partir das iterações. É interessante notar como o modelo vai aprendendo ao longo do treino e isso se reflete nas sequências inferidas a partir daqueles pesos.

Vejamos o excerto gerado após apenas 100 iterações:

Azobq. olqurma nhela e chis or Strurero ae a deele qu? l o teno o-  
eszãoenlenhetldääDmiidoiaoco indohaoi abi;llpogonde dastartes to nuemé-  
rinurase an seempisemeKr asorrrla q ée"orasalo N A vrouasuntiDo de  
qatinautelascoslenta ela cesesalEus q.i4ajaiaraa carºurÖha s e. — çes. te nuî ez.  
a erifa cJnhcigoido nuaNobê denBle,rbha.anuerintariougude e Paro m que ase,

foáe-áiéo s a emil, !iasi ôb, ene ereraia ainterte, atdha mu pos leili —xdlirade da açaar: e qáâha — altro VvaoçoirussadouaquisruquSÜimha parazIa e lóooo

O texto acima é claramente ininteligível.

Após 1000 iterações, contudo, já começamos a enxergar alguma coisa se pareça com a língua portuguesa, mas que ainda é muito problemática. Vejamos:

sei acontempor o caso, ambos no proto o pensanhho, por lar se sair ficá dizer que isso pouco seitoredo, aliás que o ravou esse. Tenho como o que broa que tembra outro de passaria mal. As afinas desdogresmos deiaros, e as criseriam na poblecaria para não vêmio que madrim as outras pratições capiais na Regininatião. — Jorga a ver prosa acunicênciia, porque foi a muridade pretencidade do ponto é do segoto de Partoni. — Ferrei essas partatérias da mãos; — deixa-me envoltar pareciu, e o pato, uma esculada por seu

A evolução continua à medida que o modelo segue treinando. Vejamos agora trechos da nossa melhor iteração:

"ter sido mais que um sorriso de almas sem madrujo. Visões proviciosa! Que pode ter morro e paterno? Quereis que o Sr. Costa e a assinato definireis tudo? Cuidamos de versos paraúnos. "Viremos ter os sentimentos de quinze e seis linhas juntos extintas"; a amparata é votata, até sempre. Entretanto, um futuro, o ato, o desejo de ficar, é padecer sem um discurso às pessoas das câmaras. Acredito também ver que eu vejo tantas coisas que o alienado não for promovido em casa, se é assim que diabo, o relógio baixo-

"por gente, pegou do marido e fê consegregar o filho para ele, e fechar-se-ia sem a falar; tinha confuso com igualdade que nos pensavam. Atalhou esta fraude ação com o seu Ablat? — Pode ser, sim, creio até em mim; eu recuei dez. E comi foi satisfeito; fecho os olhos, peguei-lhe nos lábios. Sondei que tomar à família aberta, pegando nas mãos abandonadas, onde não inclinariam algum tempo. Cinco ou meses era verdade, profundamente dinheiro, como ele mesmo pensava nos enrudoscos, mestre quando na minha opinião"

"o céu da morte, A linguagem em que passa usa o raciocanto Simão da harmonia. Ilusão desse domingo? Teve, Roma, como um pesante. A folha lá mesma e cala Feitas, ainda que ainda as moças levans à cau. Os uma feição que lhe levam tais alguns nomentos ao passado em cinco mil-réis, com outros outros e mais concertos, não menos grandes que aquele gozo secreto é sacrificado entre mim e o título sim. A observação nacional, a dona da casa, acha-se a primeira elegância do céu à nossa faca, e eu saí ao ponto de si do"

Em primeira análise, o resultado é bem melhor que os anteriores, como esperado, mas ainda muito longe de poder ser considerado um bom resultado de acordo com a normal culta da língua. Contudo, vale lembrar que o conjunto de dados é pequeno (apenas 11MB) e que também os recursos computacionais eram escassos (apenas 1 GPU). A despeito disso, o leitor atento de Machado de Assis possivelmente consiga "enxergar" um pouco do estilo do autor nos seguintes excertos gerados por nosso **Machado GPT**. Talvez possamos considerá-los como "sonhos Machadianos".

Como futuros desdobramentos deste trabalho, poderíamos tentar realizar procedimentos de *data augmentation*, ou seja, criação sintética de mais exemplos a partir da obra de Machado de Assis, com o objetivo de aumentar a base de dados. Além disso, se disponíveis os devidos recursos computacionais, poderíamos experimentar com modelos maiores, com mais parâmetros. A pergunta que fica é: dado um conjunto de dados muito rico (escrito por aquele que talvez seja considerado o maior escritor brasileiro de todos os tempos), porém pequeno para os padrões do aprendizado de máquina profundo, e dado recursos computacionais abundantes, qual o ganho de performance que seria possível obter com o pré-treinamento de um modelo como GPT?

Abaixo, apresentamos o código completo utilizado no pré-treinamento do modelo, que pode ser rodado em um notebook do Google Colab ou via Terminal, conforme explicado na página do GitHub do projeto (<https://github.com/lealmilton/machado-gpt>).

```

1 import torch
2 import torch.nn as nn
3 from torch.nn import functional as F
4 from tqdm import tqdm
5 import time
6
7 print("\n")
8 print("##### Machado-GPT #####")
9 print("\n")
10
11 # Usamos GPU se estiver disponível, caso contrário usamos CPU
12 device = "cuda" if torch.cuda.is_available() else "cpu"
13 print(f"Dispositivo de treinamento definido: {device}")
14
15
16 # Hipérparâmetros do melhor modelo obtido durante o treinamento
17 batch_size = 512 # Tamanho do lote de dados
18 context_len = 128 # Tamanho da janela de contexto
19 n_embd = 512 # Dimensão dos embeddings
20 n_head = 32 # Número de cabeças de atenção
21 n_layer = 9 # Número de blocos Transformer (Atenção + FFNN)
22 learning_rate = 1e-3 # Taxa de aprendizagem da Descida do Gradiente
23 dropout = 0.2 # Percentual de ativações ignoradas
24 eval_interval = 1000 # Intervalo de avaliações da função de perda
25 eval_iters = 50 # Número de iterações usadas para o cálculo da perda
26 max_iters = 10000 # Número total de iterações do treinamento
27
28 # Leitura dos dados concatenados
29 print("\nLendo os dados de treinamento...")
30 text = open("processado/textos_concatenados.txt", "r", encoding="utf-8").
31     read()
32
33 # Criação do vocabulário composto de caracteres únicos
34 chars = sorted(set(text))
35 vocab_size = len(chars)
36 print("\nNúmero de tokens do vocabulário:", vocab_size)
37
38 # Indexação dos tokens
39 char_to_idx = {ch: i for i, ch in enumerate(chars)}
40 idx_to_char = {i: ch for i, ch in enumerate(chars)}
```

```

40
41 # Funções para codificar e decodificar índices e tokens
42 def encode(text):
43     return [char_to_idx[ch] for ch in text]
44
45 def decode(indices):
46     return "".join(idx_to_char[i] for i in indices)
47
48 # Codificação em índices dos dados de treinamento
49 data = torch.tensor(encode(text), dtype=torch.long)
50 print("\nNúmero total de tokens no conjunto de dados:", len(data))
51
52 print("\nDividindo os dados em conjuntos de treino e validação...")
53
54 # Divisão dos dados em treino e validação
55 train_percent = 0.9
56 n = int(train_percent * len(data))
57 train_data = data[:n]
58 val_data = data[n:]
59
60 # Função para obter lotes de dados
61 def get_batch(split):
62     # Seleciona se os dados virão do conjunto de treino ou validação
63     data = train_data if split == "train" else val_data
64     # Gera batch_size sequências aleatórias de índices iniciais para cada
65     # lote
66     ix = torch.randint(len(data) - context_len, (batch_size,))
67     # Gera sequências de treinamento baseadas no tamanho da janela de
68     # contexto
69     x = torch.stack([data[i : i + context_len] for i in ix])
70     # Gera rótulos para as sequências de treinamento
71     y = torch.stack([data[i + 1 : i + context_len + 1] for i in ix])
72     # Envia os tensores para a GPU ou CPU
73     x, y = x.to(device), y.to(device)
74     return x, y
75
76 # Decorador para evitar o cálculo dos gradientes durante o cálculo da perda
77 @torch.no_grad()
78 # Função que calcula a perda
79 def estimate_loss():
80     out = []
81     # Coloca o modelo em modo de avaliação
82     model.eval()
83     # Itera sobre os conjuntos de dados
84     for split in ["train", "val"]:
85         # Inicializa o vetor de perdas com zeros
86         losses = torch.zeros(eval_iters)
87         # Itera sobre o número de avaliações pré-determinadas
88         for k in range(eval_iters):
89             # Obtém exemplos aleatórios de sequências dos conjuntos de dados
90             X, Y = get_batch(split)
91             # Obtém os logits e a perda para os exemplos selecionados
92             logits, loss = model(X, Y)
93             # Atualiza o vetor de perdas
94             losses[k] = loss.item()
95
96     # Calcula a média da perda para cada conjunto de dados

```

```

94         out[split] = losses.mean()
95     # Recoloca o modelo em modo de treinamento
96     model.train()
97     return out
98
99 # Define a função que calcula a codificação posicional
100 def positional_encoding(context_len, n_embd, device):
101     # Gera um tensor de posições
102     position = torch.arange(context_len,
103                             dtype=torch.float32,
104                             device=device).unsqueeze(1)
105     # Calcula o termo divisor para as funções seno e cosseno
106     div_term = torch.exp(
107         torch.arange(0, n_embd, 2, device=device).float() *
108         (-torch.log(torch.tensor(10000.0, device=device)) / n_embd))
109     # Inicializa o tensor de codificação posicional com zeros
110     pe = torch.zeros(context_len, n_embd, device=device)
111     # Aplica a função seno aos índices pares da matriz posicional
112     pe[:, 0::2] = torch.sin(position * div_term)
113     # Aplica a função cosseno aos índices ímpares da matriz posicional
114     pe[:, 1::2] = torch.cos(position * div_term)
115     return pe
116
117 # Define a classe para uma cabeça de atenção
118 class Head(nn.Module):
119     def __init__(self, head_size):
120         super().__init__()
121         # Inicializa as matrizes de chaves, consultas e valores
122         # Não inicializamos os viéses para simplificar o modelo
123         self.key = nn.Linear(n_embd, head_size, bias=False)
124         self.query = nn.Linear(n_embd, head_size, bias=False)
125         self.value = nn.Linear(n_embd, head_size, bias=False)
126         # Registra um buffer para a máscara de atenção triangular inferior
127         # Trata-se de matriz com 1's abaixo da diagonal principal (inclusive)
128         # e zeros acima. O uso de 'register_buffer' permite que o tensor 'tril',
129
130             # seja movido junto com o modelo para a GPU, se disponível, e
131             # não seja considerado um parâmetro do modelo.
132             self.register_buffer("tril", torch.tril(
133                 torch.ones(context_len, context_len)
134             ))
135             # Inicializa a camada de Dropout
136             self.dropout = nn.Dropout(dropout)
137
138     # Define a passagem forward do modelo
139     def forward(self, x):
140         # Obtém as dimensões do conjunto de dados
141         batch_size, context_len, n_embd = x.shape
142         # Projeção linear da entrada através da matriz de pesos das chaves
143         k = self.key(x)
144         # Projeção linear da entrada através da matriz de pesos das consultas
145         q = self.query(x)
146         # Produto escalar entre q e k com dimensionamento sqrt(n_embd)
147         wei = q @ k.transpose(-2, -1) * n_embd ** (-0.5)
148         # Aplicação da máscara na matriz resultado da operação anterior
149         # Note que as entradas iguais a zero são redefinidas como -inf

```

```

149     wei = wei.masked_fill(
150         self.tril[:context_len, :context_len] == 0, float("-inf")
151     )
152     # Aplicação da função Softmax
153     wei = F.softmax(wei, dim=-1)
154     # Aplicação do Dropout
155     wei = self.dropout(wei)
156     # Projeção linear da entrada através da matriz de pesos dos valores
157     v = self.value(x)
158     # Multiplicação final pela matriz de valores
159     out = wei @ v
160     return out
161
162 # Definição da classe de múltiplas cabeças
163 class MultiHeadAttention(nn.Module):
164     def __init__(self, num_head, head_size):
165         super().__init__()
166         # Inicializa uma lista de módulos
167         # com várias instâncias da cabeça de atenção
168         self.heads = nn.ModuleList([Head(head_size) for _ in range(num_head)])
169         # Inicializa uma camada linear de projeção dos dados
170         self.proj = nn.Linear(n_embd, n_embd)
171         # Inicializa a camada de Dropout
172         self.dropout = nn.Dropout(dropout)
173     # Define a passagem forward do modelo
174     def forward(self, x):
175         # Aplica as cabeças de atenção sobre os dados
176         # e as concatena no final
177         out = torch.cat([h(x) for h in self.heads], dim=-1)
178         # Realiza a projeção linear do resultado anterior
179         # e aplica o dropout em seguida
180         out = self.dropout(self.proj(out))
181     return out
182
183 # Define a classe da rede Feed Forward
184 class FeedForward(nn.Module):
185     def __init__(self, n_embd):
186         super().__init__()
187         # Cria a arquitetura da rede com uma camada linear,
188         # seguida de uma ativação ReLU, seguida por outra
189         # camada linear e, por fim, com uma camada de dropout
190         self.net = nn.Sequential(
191             nn.Linear(n_embd, 4 * n_embd),
192             nn.ReLU(),
193             nn.Linear(4 * n_embd, n_embd),
194             nn.Dropout(dropout),
195         )
196     # Define a passagem forward do modelo
197     def forward(self, x):
198         # Aplica a sequência de camadas definida para a FFNN
199         return self.net(x)
200
201 # Define um bloco Transformer
202 class Block(nn.Module):
203     def __init__(self, n_embd, n_head):
204         super().__init__()

```

```

205     # Determina o tamanho de cada cabeça de atenção
206     head_size = n_embd // n_head
207     # Inicializa as múltiplas cabeças de atenção
208     self.sa = MultiHeadAttention(n_head, head_size)
209     # Inicializa a camada FFNN
210     self.ffwd = FeedForward(n_embd)
211     # Inicializa as camadas de normalização
212     self.ln1 = nn.LayerNorm(n_embd)
213     self.ln2 = nn.LayerNorm(n_embd)
214
215     # Define a passagem forward do modelo
216     def forward(self, x):
217         # Computa a normalização dos dados, seguida pela atenção com
218         # múltiplas cabeças, além da soma da conexão residual
219         x = x + self.sa(self.ln1(x))
220         # Computa a normalização dos dados, seguida pela FFNN,
221         # além da soma da conexão residual
222         x = x + self.ffwd(self.ln2(x))
223         return x
224
225     # Define a classe do modelo GPT
226     class GPTLanguageModel(nn.Module):
227         def __init__(self):
228             super().__init__()
229             # Inicializa a matriz de pesos dos embeddings dos tokens
230             self.token_embedding = nn.Embedding(vocab_size, n_embd)
231             # Gera e armazena embeddings posicionais
232             self.positional_embeddings = positional_encoding(
233                 context_len, n_embd, device
234             )
235             # Cria uma sequência de tamanho n_layer de blocos Transformer
236             self.blocks = nn.Sequential(
237                 *[Block(n_embd, n_head=n_head) for _ in range(n_layer)]
238             )
239             # Inicializa a camada de normalização
240             self.ln_f = nn.LayerNorm(n_embd)
241             # Inicializa a camada linear
242             self.lm_head = nn.Linear(n_embd, vocab_size)
243
244         # Define a passagem forward do modelo
245         def forward(self, idx, targets=None):
246             # Obtém as dimensões do conjunto de dados
247             batch_size, context_len = idx.shape
248             # Embeddings de tokens
249             tok_emb = self.token_embedding(idx)
250             # Codificações posicionais dos tokens
251             pos_emb = self.positional_embeddings[:context_len, :]
252             # Soma os embeddings às codificações posicionais
253             x = tok_emb + pos_emb
254             # Aplica os Blocos Transformer nos dados
255             x = self.blocks(x)
256             # Aplica a camada de normalização
257             x = self.ln_f(x)
258             # Aplica a camada linear final para obtenção dos logits
259             logits = self.lm_head(x)
260

```

```

261     # Calcula a perda se os rótulos forem fornecidos
262     if targets is None:
263         loss = None
264     else:
265         # Obtém as dimensões dos logits
266         batch_size, context_len, n_embd = logits.shape
267         # Redimensiona (flattens) os 'logits' para uma matriz 2D,
268         # onde a primeira dimensão é batch_size * context_len e
269         # a segunda é n_embd .Isso é necessário para cálculo da
270         # função de perda de entropia cruzada
271         logits = logits.view(batch_size * context_len, n_embd)
272         # Redimensiona os rótulos alvo para um vetor 1D com tamanho
273         # batch_size * context_len para corresponder
274         # aos 'logits' redimensionados
275         targets = targets.view(batch_size * context_len)
276         # Calcula a perda usando a entropia cruzada, que compara
277         # os 'logits' da rede com os rótulos alvo.
278         loss = F.cross_entropy(logits, targets)
279
280     return logits, loss
281
282     # Gera novas sequências de texto
283 def generate(self, idx, max_new_tokens):
284     # Itera até o tamanho máximo de tokens a ser gerado
285     for _ in range(max_new_tokens):
286         # Limita a entrada ao tamanho da janela de contexto
287         idx_cond = idx[:, -context_len:]
288         # Calcula os logits para o contexto de entrada
289         logits, loss = self(idx_cond)
290         # Extrai os logits
291         logits = logits[:, -1, :]
292         # Calcula as probabilidades com a Softmax
293         probs = F.softmax(logits, dim=-1)
294         # Sorteia o próximo token a partir de uma distribuição
295         # multinomial. Aqui, obtemos apenas uma amostra (token)
296         idx_net = torch.multinomial(probs, num_samples=1)
297         # Concatena o novo token ao contexto de entrada
298         idx = torch.cat((idx, idx_net), dim=1)
299     return idx
300
301 def train():
302     start = time.time()
303
304     # Barra de progresso do treinamento
305     progress_bar = tqdm(total=max_iters, desc="Progresso do treinamento")
306
307     # Loop de treinamento para max_iters iterações
308     for iter in range(max_iters):
309         # Verifica se é hora de estimar a perda e exibir informações
310         if iter % eval_interval == 0 or iter == max_iters-1:
311             # Calcula e exibe as perdas
312             losses = estimate_loss()
313
314             print(f"\nIteração {iter}:", f"Perda de Treino {losses['train']:.4f}", f"Perda de Validação {losses['val']:.4f}")
315
316

```

```

317
318     # Gera novo texto a partir do modelo
319     context = torch.zeros((1, 1), dtype=torch.long, device=device)
320     output_gerado = decode(
321         model.generate(context, max_new_tokens=512)[0].tolist()
322     )
323     print(f"\nTexto gerado pelo modelo na iteração: {iter}")
324     print(output_gerado)
325
326     # Salva checkpoint do modelo
327     caminho_salvamento_modelo = f"modelo_{iter}.pth"
328     torch.save(model.state_dict(), caminho_salvamento_modelo)
329
330     # Obtém lotes de dados de treinamento
331     xb, yb = get_batch("train")
332     # Realiza o passo à frente do modelo, calculando os logits e loss
333     logits, loss = model(xb, yb)
334     # Zera os gradientes dos parâmetros do modelo
335     optimizer.zero_grad(set_to_none=True)
336     # Calcula os gradientes da perda em relação aos parâmetros
337     # Backpropagation
338     loss.backward()
339     # Atualiza os parâmetros do modelo usando o otimizador
340     optimizer.step()
341
342     # Atualiza barra de progresso
343     progress_bar.update(1)
344
345     # Encerra barra de progresso
346     progress_bar.close()
347
348     end = time.time()
349     total_time = end - start
350     print(f"\nTempo total de treinamento em segundos: {float(total_time):.2f}")
351     print("\n")
352
353     # Inicializa o modelo
354     model = GPTLanguageModel()
355     # Move o modelo para o dispositivo apropriado (GPU/CPU)
356     m = model.to(device)
357     # Configura o otimizador com taxa de aprendizagem específica
358     optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
359
360     # Calcula o número de parâmetros
361     total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
362     print(f"\nNúmero total de parâmetros: {total_params}")
363
364     print("\nInício do processo de treinamento...\n")
365     train()
366     print("\nTreinamento concluído!")
367
368     print("\n##### Fim do programa #####")

```

# Referências

- [ANDREJ KARPATHY 2023] ANDREJ KARPATHY. *Let's build GPT: from scratch, in code, spelled out*. Fev. de 2023. URL: <https://www.youtube.com/watch?v=kCc8FmEb1nY> (citado na pg. 4).
- [BA *et al.* 2016] Jimmy Lei BA, Jamie Ryan KIROS e Geoffrey E. HINTON. “Layer Normalization” (jul. de 2016) (citado na pg. 41).
- [BAHDANAU *et al.* 2014] Dzmitry BAHDANAU, Kyunghyun CHO e Yoshua BENGIO. “Neural Machine Translation by Jointly Learning to Align and Translate” (set. de 2014) (citado na pg. 10).
- [L. BAHL *et al.* 1974] L. BAHL, J. COCKE, F. JELINEK e J. RAVIV. “Optimal decoding of linear codes for minimizing symbol error rate (Corresp.)” *IEEE Transactions on Information Theory* 20.2 (mar. de 1974), pp. 284–287. ISSN: 0018-9448. DOI: [10.1109/TIT.1974.1055186](https://doi.org/10.1109/TIT.1974.1055186) (citado na pg. 8).
- [L. R. BAHL *et al.* 1983] Lalit R. BAHL, Frederick JELINEK e Robert L. MERCER. “A Maximum Likelihood Approach to Continuous Speech Recognition”. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5.2 (mar. de 1983), pp. 179–190. ISSN: 0162-8828. DOI: [10.1109/TPAMI.1983.4767370](https://doi.org/10.1109/TPAMI.1983.4767370) (citado na pg. 8).
- [BENGIO *et al.* 2003] Yoshua BENGIO, Réjean DUCHARME, Pascal VINCENT e Christian JAUVIN. “A Neural Probabilistic Language Model”. *CrossRef Listing of Deleted DOIs* 1.6 (2003). ISSN: 0003-6951. DOI: [10.1162/153244303322533223](https://doi.org/10.1162/153244303322533223). URL: [http://www.crossref.org/deleted\\_DOI.html](http://www.crossref.org/deleted_DOI.html) (citado nas pgs. 8, 9, 22).
- [BROWN *et al.* 2020] Tom B. BROWN *et al.* “Language Models are Few-Shot Learners”. abs/2005.14165 (mai. de 2020). URL: <http://arxiv.org/abs/2005.14165> (citado nas pgs. 1, 11, 12).
- [BUBECK *et al.* 2023] Sébastien BUBECK *et al.* “Sparks of Artificial General Intelligence: Early experiments with GPT-4” (mar. de 2023). URL: <http://arxiv.org/abs/2303.12712> (citado na pg. 1).
- [CHOMSKY 1957] Noam CHOMSKY. *Syntactic Structures*. De Gruyter, dez. de 1957. ISBN: 9783112316009. DOI: [10.1515/9783112316009](https://doi.org/10.1515/9783112316009) (citado na pg. 7).

## REFERÊNCIAS

- [COLLOBERT e WESTON 2008] Ronan COLLOBERT e Jason WESTON. “A unified architecture for natural language processing”. In: *Proceedings of the 25th international conference on Machine learning - ICML '08*. New York, New York, USA: ACM Press, 2008, pp. 160–167. ISBN: 9781605582054. DOI: [10.1145/1390156.1390177](https://doi.org/10.1145/1390156.1390177) (citado na pg. 9).
- [DEVLIN *et al.* 2018] Jacob DEVLIN, Ming-Wei CHANG, Kenton LEE e Kristina TOUTANOVA. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” (out. de 2018) (citado na pg. 2).
- [HAO LI 2018] HAO LI. “Visualizing the Loss Landscape of Neural Nets” (2018) (citado na pg. 43).
- [HE *et al.* 2015] Kaiming HE, Xiangyu ZHANG, Shaoqing REN e Jian SUN. “Deep Residual Learning for Image Recognition” (dez. de 2015) (citado na pg. 42).
- [HOCHREITER e SCHMIDHUBER 1997] Sepp HOCHREITER e Jürgen SCHMIDHUBER. “Long Short-Term Memory”. *Neural Computation* 9.8 (nov. de 1997), pp. 1735–1780. ISSN: 08997667. DOI: [10.1162/neco.1997.9.8.1735](https://doi.org/10.1162/neco.1997.9.8.1735) (citado na pg. 8).
- [IOFFE e SZEGEDY 2015] Sergey IOFFE e Christian SZEGEDY. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” (fev. de 2015). URL: <http://arxiv.org/abs/1502.03167> (citado na pg. 41).
- [JAKOB USZKOREIT 2017] JAKOB USZKOREIT. *Transformer: A Novel Neural Network Architecture for Language Understanding – Google AI Blog*. 2017. URL: <https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html> (citado na pg. 1).
- [JURAFSKY e MARTIN 2009] Daniel JURAFSKY e JH MARTIN. *Speech & language processing*. 5. 2009. ISBN: 9780131873216 (citado na pg. 17).
- [KAMATH *et al.* 2022] Uday KAMATH, Kenneth L. GRAHAM e Wael EMARA. *Transformers for Machine Learning*. Chapman & Hall/CRC Machine Learning & Pattern Recognition. Boca Raton: Chapman e Hall/CRC, abr. de 2022. ISBN: 9781003170082. DOI: [10.1201/9781003170082](https://doi.org/10.1201/9781003170082). URL: <https://www.taylorfrancis.com/books/9781003170082> (citado na pg. 2).
- [LIU *et al.* 2018] Peter J. LIU *et al.* “Generating Wikipedia by Summarizing Long Sequences” (jan. de 2018) (citado nas pgs. 1, 11).
- [MIKOLOV *et al.* 2013] Tomas MIKOLOV, Kai CHEN, Greg CORRADO e Jeffrey DEAN. “Efficient Estimation of Word Representations in Vector Space”. *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings* (jan. de 2013). URL: <http://arxiv.org/abs/1301.3781> (citado nas pgs. 9, 22).

## REFERÊNCIAS

- [MUKUL RATHI 2018] MUKUL RATHI. *Demystifying Deep Learning*. 2018. URL: <https://mukulrathi.com/demystifying-deep-learning/feed-forward-neural-network/> (citado na pg. 38).
- [OPENAI 2023] OPENAI. “GPT-4 Technical Report” (mar. de 2023). URL: <http://arxiv.org/abs/2303.08774> (citado nas pgs. 1, 13, 14).
- [OUYANG *et al.* 2022] Long OUYANG *et al.* “Training language models to follow instructions with human feedback” (mar. de 2022) (citado nas pgs. 1, 12, 53, 54).
- [RADFORD, NARASIMHAN *et al.* 2018] Alec RADFORD, Karthik NARASIMHAN, Tim SALIMANS e Ilya SUTSKEVER. *Improving Language Understanding by Generative Pre-Training*. 2018 (citado nas pgs. 1, 11).
- [RADFORD, WU *et al.* 2019] Alec RADFORD, Jeffrey WU *et al.* “Language Models are Unsupervised Multitask Learners” (2019). URL: <https://github.com/codelucas/newspaper> (citado nas pgs. 1, 11).
- [RONG 2014] Xin RONG. “word2vec Parameter Learning Explained” (nov. de 2014) (citado na pg. 27).
- [ROTHMAN 2021] D ROTHMAN. *Transformers for Natural Language Processing: Build Innovative Deep Neural Network Architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and More*. Packt Publishing, 2021. ISBN: 9781800565791. URL: <https://books.google.com.br/books?id=Ua03zgEACAAJ> (citado na pg. 2).
- [SEBASTIAN RASCHKA 2023] SEBASTIAN RASCHKA. *LM Training: RLHF and Its Alternatives*. Set. de 2023. URL: <https://magazine.sebastianraschka.com/p/llm-training-rlhf-and-its-alternatives> (citado na pg. 4).
- [SENNRICH *et al.* 2016] Rico SENNRICH, Barry HADDOW e Alexandra BIRCH. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2016, pp. 1715–1725. DOI: [10.18653/v1/P16-1162](https://doi.org/10.18653/v1/P16-1162) (citado na pg. 17).
- [SHANNON 1951] C. E. SHANNON. “Prediction and Entropy of Printed English”. *Bell System Technical Journal* 30.1 (jan. de 1951), pp. 50–64. ISSN: 00058580. DOI: [10.1002/j.1538-7305.1951.tb01366.x](https://doi.org/10.1002/j.1538-7305.1951.tb01366.x) (citado na pg. 6).
- [SRIVASTAVA *et al.* 2014] Nitish SRIVASTAVA, Geoffrey HINTON, Alex KRIZHEVSKY e Ruslan SALAKHUTDINOV. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Rel. técn. 2014, pp. 1929–1958 (citado na pg. 43).
- [TURING 1950] A. M. TURING. “I.—COMPUTING MACHINERY AND INTELLIGENCE”. *Mind* LIX.236 (out. de 1950), pp. 433–460. ISSN: 1460-2113. DOI: [10.1093/mind/LIX.236.433](https://doi.org/10.1093/mind/LIX.236.433) (citado na pg. 7).

## REFERÊNCIAS

- [VASWANI *et al.* 2017] Ashish VASWANI *et al.* “Attention Is All You Need” (jun. de 2017). URL: <http://arxiv.org/abs/1706.03762> (citado nas pgs. 1, 2, 10, 28, 33–35, 42).
- [WANG *et al.* 2019] Bin WANG, Angela WANG, Fenxiao CHEN, Yuncheng WANG e C.-C. Jay Kuo. “Evaluating word embedding models: methods and experimental results”. *APSIPA Transactions on Signal and Information Processing* 8.1 (2019). ISSN: 2048-7703. doi: [10.1017/AT SIP.2019.12](https://doi.org/10.1017/AT SIP.2019.12). URL: <http://www.nowpublishers.com/article/Details/SIP-124> (citado na pg. 22).
- [WEIZENBAUM 1966] Joseph WEIZENBAUM. “ELIZA—a computer program for the study of natural language communication between man and machine”. *Communications of the ACM* 9.1 (jan. de 1966), pp. 36–45. ISSN: 0001-0782. doi: [10.1145/365153.365168](https://doi.org/10.1145/365153.365168) (citado na pg. 7).
- [ZHUO *et al.* 2023] Terry Yue ZHUO, Yujin HUANG, Chunyang CHEN e Zhenchang XING. “Red teaming ChatGPT via Jailbreaking: Bias, Robustness, Reliability and Toxicity” (jan. de 2023) (citado na pg. 13).