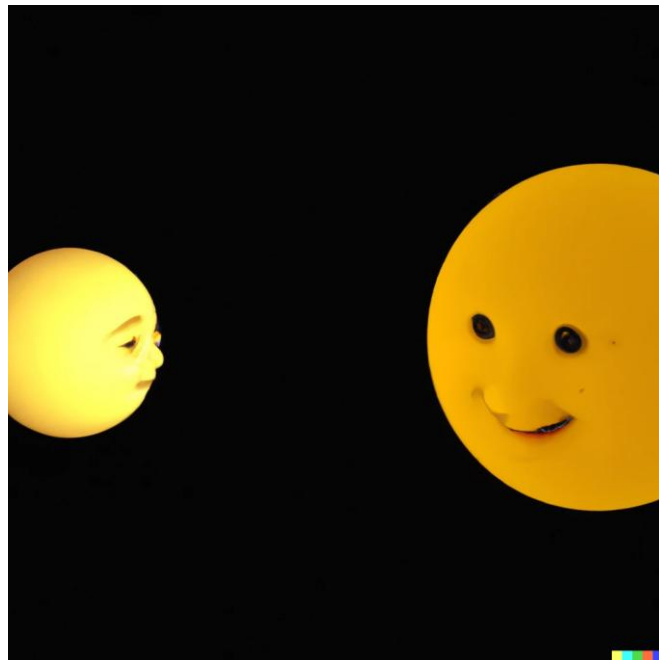


# ***Conversa entre o Sol e Plutão***

***Treinando modelos de linguagem com redes neurais Transformer***



**Márcio Galvão**  
galva.marcio@gmail.com

Este texto é compartilhado pelo autor com fins puramente educativos, sem propósitos comerciais, através de uma licença CC-NC-4.0.



[Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

*Imagem da capa ("Sol conversando com Plutão") gerada pelo autor com o DALL.E2 da OpenAI*

Alguns conteúdos são traduzidos livremente ou são adaptações de diferentes fontes, sempre citadas com o devido crédito. As imagens utilizadas são liberadas para uso (*free*) ou têm suas fontes devidamente creditadas. Nomes de produtos mencionados são marcas registradas das respectivas empresas. A menção a produtos ou serviços é apenas para fins informativos e não constitui um endosso ou recomendação.

O autor se esforçará para manter o texto atualizado em futuras revisões, mas não há garantias de que todo o conteúdo esteja completamente atualizado em qualquer instante. O conteúdo é compartilhado de boa fé e o autor não assume responsabilidade sobre sua utilização dado que *não é especialista* no assunto e pode ter feito erros de interpretação, de modo que é conveniente que as informações aqui fornecidas sejam confirmadas em outras fontes.

O autor agradece à Módulo Security Solutions (<https://www.modulo.com.br/>) pelo financiamento das pesquisas que tornaram este trabalho possível.

Caso você queira colaborar com sugestões de referências, críticas ou apontar erros neste material por favor escreva para [galva.marcio@gmail.com](mailto:galva.marcio@gmail.com). Toda colaboração é bem-vinda e será devidamente apreciada.

Versão 1.5 em 09 de janeiro de 2024

Márcio Galvão

Revisão: Alberto Bastos ([abastos@modulo.com](mailto:abastos@modulo.com))

*Para o amigo e mestre*

***Fernando Barcellos Ximenes***

# Conteúdo

## Introdução

Apresentação  
Sobre o ChatGPT  
Que tarefas de NLP o ChatGPT é capaz de executar?

## Capítulo 1- Modelos de Linguagem

- 1.1. Processamento de Linguagem Natural (NLP - *Natural Language Processing*)
- 1.2. A evolução dos Modelos de Linguagem
  - 1.2.1. Tamanho de modelo não é documento
  - 1.2.2. Modelos genéricos e específicos
  - 1.2.3. Modelos de texto e modelos multimodais
  - 1.2.4. LLMs textuais completam textos
  - 1.2.5. MaaS (*Model-as-a-Service*)
  - 1.2.6. Customizando modelos de linguagem

## Capítulo 2 - Tokenização e *Embeddings*

- 2.1. Semântica Distribucional
- 2.2. Modelos de Linguagem Probabilísticos
- 2.3. Modelos de Linguagem Neurais
- 2.4. De onde vêm as probabilidades?
- 2.5. Tokenização
- 2.6. *Word Embeddings*
- 2.7. Qual é o próximo *token* na sequência?
- 2.8. Amostragem softmax
- 2.9. Modelo de Linguagem autoregressivo
- 2.10. De onde vem tanto texto?
- 2.11. Diversidade nas respostas
- 2.12. ELMO, BERT e GPT

## Capítulo 3 - Treinando o ChatGPT

- 3.1. Pipeline de treinamento de Assistentes GPT
  - Estágio 1 - Pré-treinamento
  - Estágio 2 - Ajuste Supervisionado via *Prompts*
  - Estágio 3 - Modelo de Recompensas (RM)
  - Estágio 4 - Aprendizado com Reforço
- 3.2. Vantagens e limitações dos modelos RLHF
- 3.3. Se quer respostas mais precisas, peça por isso
- 3.4. Para tarefas mais especializadas, use *plugins*

## Capítulo 4 - Atenção é tudo o que você precisa

- 4.1. Redes Neurais
- 4.2. Redes Neurais Recorrentes (RNNs)
- 4.3. A caixa mágica
- 4.4. O mecanismo de Atenção
- 4.5. A arquitetura *Transformer*
  - 4.5.1. ENCODERS
    - 4.5.1.1. Codificação posicional dos vetores de *Embeddings*
    - 4.5.1.2. Vetorização
    - 4.5.1.3. O cálculo da Atenção
    - 4.5.1.4. Matrizes em vez de vetores
    - 4.5.1.5. A subcamada FFNN no ENCODER

4.5.1.6. Para que tantas camadas de ENCODERS?

4.5.1.7. Atenção *Multi-Head*

4.5.1.8. Conexões residuais e camada de normalização

### 4.5.2. DECODERS

4.5.2.1. Decodificação passo a passo

4.5.2.2. Soma de saídas e normalização

4.5.2.3. A subcamada FFNN nos DECODERS

4.5.2.4. Camada Linear e camada Softmax

### 4.6. Revisão

## Conclusão e referências

Conclusão

Referências

# Introdução

## Apresentação

Este texto trata do treinamento dos "grandes modelos de linguagem" (*LLMs - Large Language Models*) pelas redes neurais com arquitetura *Transformer*, no contexto do Processamento da Linguagem Natural (*NLP - Natural Language Processing*), uma das mais importantes categorias do Aprendizado de Máquina (*Machine Learning*).

Além da **Introdução** e da **Conclusão**, o texto é dividido em quatro partes:

- O **Capítulo 1** apresenta os modelos de linguagem (*LMs - Language Models*) - o que são, para que servem, como podem ser categorizados e como evoluíram ao longo dos últimos anos.
- O **Capítulo 2** trata de tokenização e "*Word Embedding*", técnicas importantes no treinamento de LMs.
- O **Capítulo 3** explica como são treinados os modelos GPT (*Generative Pre-Trained Transformer*) da OpenAI.
- O **Capítulo 4** discute de forma introdutória as redes neurais com arquitetura *Transformer*, tecnologias fundamentais no treinamento de LLMs e outros modelos de IA Generativa, com ênfase nos mecanismos de Atenção.

O texto é destinado a consultores, executivos, desenvolvedores, pesquisadores e outros interessados em IA Generativa, no ChatGPT e nas redes neurais *Transformer*.

Como o texto é introdutório, são fornecidas muitas referências para os que desejarem se aprofundar. Alguns conteúdos mais técnicos foram separados do texto principal em BOXES para os leitores mais interessados.

## Sobre o ChatGPT

O ChatGPT [1] é um Assistente de IA que permite ao usuário interagir com diferentes versões de modelos de linguagem, e alcançou rapidamente enorme popularidade após seu lançamento pela OpenAI [2] em 2022. Existe vasta literatura sobre como o ChatGPT pode ser acessado e utilizado, de modo que estaremos mais direcionados a entender *como ele funciona*, ou mais precisamente, *como são treinados* os grandes modelos de linguagem (LLM - *Large Language Models*) que o ChatGPT utiliza "por trás dos panos".

O ChatGPT é um Assistente de IA (Inteligência Artificial). Podemos pensar nele como um *chatbot* de propósito geral capaz de responder perguntas em domínio aberto (sobre qualquer assunto), bem como executar outras tarefas de processamento de linguagem natural como fazer traduções de idiomas e correções gramaticais. O Assistente também é capaz (com limitações) de executar operações aritméticas simples, resolver problemas envolvendo raciocínio científico ou lógico, e até mesmo gerar código de computador em diversas linguagens. Entretanto, como veremos, apesar de toda esta flexibilidade para executar diferentes tipos de tarefas o que o ChatGPT *realmente* faz é **completar textos** - ou seja, gerar novos conteúdos sintéticos.

As "respostas" para todas estas tarefas ou perguntas são geradas a partir de instruções passadas pelos usuários (*Prompts*) em linguagem natural pela interfaces do aplicativo para a Web (navegador) ou para Smartphones (iOs ou Android). Embora suporte diversos idiomas, a performance é melhor (mais consistente) em Inglês. O ChatGPT utiliza (atualmente) dois modelos de linguagem generativos com arquitetura GPT (*Generative Pre-Trained Transformer*) previamente treinados pela OpenAI. O modelo GPT-3.5 [4] (uma evolução do GPT-3 [3]) pode ser acessado gratuitamente. Já o modelo GPT-4 [5], bem mais poderoso, requer assinatura do plano ChatGPT Plus que é pago. Estes modelos que suportam o ChatGPT (em diferentes versões) também podem ser acessados diretamente como serviços via API, e assim podem ser integrados com aplicações corporativas (*MaaS - Model-as-a-Service* - ver Seção 1.2.5).

No Capítulo 3 veremos como é feito o treinamento dos modelos de linguagem GPT da OpenAI. No caso dos modelos mais sofisticados (modelos *RLHF* - Seção 3.2), o treinamento é realizado por redes neurais com arquitetura *Transformer*. O processo ocorre em quatro estágios sequenciais.

- O primeiro estágio é não supervisionado, consome quantidades massivas de dados da Internet e requer supercomputadores agregando milhares de GPUs.
- Em seguida, são executados outros três estágios onde o esforço computacional é menor, e são utilizados menos dados com maior qualidade, e o treinamento é realizado com supervisão humana.
- No total, o processo completo pode demorar meses e os custos podem atingir milhões de dólares.

Uma vez treinado o modelo de linguagem que vai utilizar, o ChatGPT é capaz de gerar seus conteúdos sintéticos (textos) em resposta às instruções (*Prompts*) fornecidas pelos usuários *em tempo real*. Em geral, as respostas do ChatGPT são geradas em questão de segundos. No entanto, o tempo exato pode variar com a complexidade do *Prompt*, o comprimento da resposta desejada, a carga de trabalho nos servidores da OpenAI e de outros fatores dinâmicos.

Como veremos através de vários exemplos ao longo do texto, a construção adequada destes *Prompts* pode fazer grande diferença na qualidade (precisão, relevância) das respostas obtidas, seja qual for a natureza da tarefa de linguagem em questão (completar textos, traduzir idiomas, analisar sentimentos, cálculos aritméticos, raciocínio lógico etc.). Não temos espaço para discutir aqui a *Engenharia de Prompts* com a profundidade que o assunto merece, mas algumas referências e recomendações são fornecidas Seção 3.3.

Para executar com maior precisão uma maior variedade de tarefas o ChatGPT pode ser integrado a outros aplicativos mais adequados para realizar funções especializadas (por exemplo, calculadoras, para fazer cálculos aritméticos). Tal integração pode ocorrer através de *plugins* como explicado na Seção 3.4.



O ChatGPT "levou a IA para as massas", ao permitir que pessoas comuns (no sentido de "não especialistas") interagissem com relativa facilidade com uma aplicação de IA altamente sofisticada, usando um computador ou o telefone celular com acesso à Internet.

A excelente performance demonstrada (em media) pelo ChatGPT em diferentes domínios do conhecimento e em variados tipos de tarefas causou perplexidade até nos pesquisadores - muitas pessoas ficaram com medo de perder seus empregos para o Assistente, que se tornou *a aplicação mais popular e de crescimento mais rápido da história* [6].

*Figura 1 - O ChatGPT alcançou um milhão de usuários em apenas cinco dias (o Instagram demorou dois meses e meio), e já contava com mais de cem milhões de usuários em fevereiro de 2023.*

## ChatGPT Sprints to One Million Users

Time it took for selected online services to reach one million users



\* one million backers \*\* one million nights booked \*\*\* one million downloads

Source: Company announcements via Business Insider/LinkedIn



statista

A OpenAI recebeu mais de U\$ 13 bilhões em investimentos da Microsoft desde 2019, e como parte do acordo utiliza sua infraestrutura de computação em nuvem (o Microsoft Azure) [7], que fornece a imensa escalabilidade (computação de alta performance, armazenamento, *networking*, segurança etc.) necessária tanto para treinar grandes modelos de linguagem quanto para hospedar aplicações de IA Generativa como o ChatGPT e outras. Estas aplicações raramente estão fora do ar, e atendem com bom desempenho aos acessos simultâneos de milhões de usuários em todo o mundo.

Para utilizar o ChatGPT basta criar uma conta na OpenAI [8], e em seguida fazer o *Log in* no serviço (Figura 2). Mencionamos que o ChatGPT tem uma versão gratuita que utiliza o modelo GPT 3.5 [4], e também uma versão paga baseada no modelo GPT-4 [5] (Figura 3). Este modelo foi lançado em 2023 pela OpenAI, e estabeleceu um novo patamar em termos de desempenho e quantidade de parâmetros no treinamento (detalhes sobre o GPT-4 ainda não divulgados).

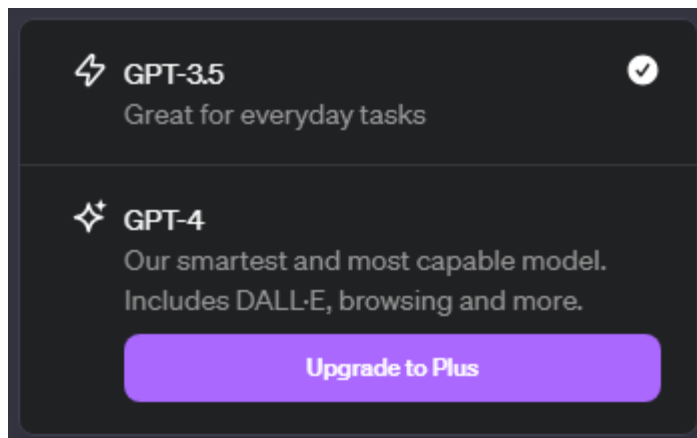



Figura 3

## Create your account

Continue

Already have an account? [Log in](#)

OR

 Continue with Google

 Continue with Microsoft Account


 Continue with Apple

Figura 2

A Figura 4 mostra uma interação simples do autor com o ChatGPT, com uma pergunta científica. Observe que no *Prompt* o Assistente foi orientado a "pensar como um especialista" e a "responder com máxima precisão". A justificativa para isso será explicada na Seção 3.3.

Apesar de sua boa performance (de modo geral), o ChatGPT não de forma alguma infalível, como alerta a própria OpenAI na mensagem exibida na parte inferior da interface:

"ChatGPT pode cometer erros. Considere verificar informações importantes".

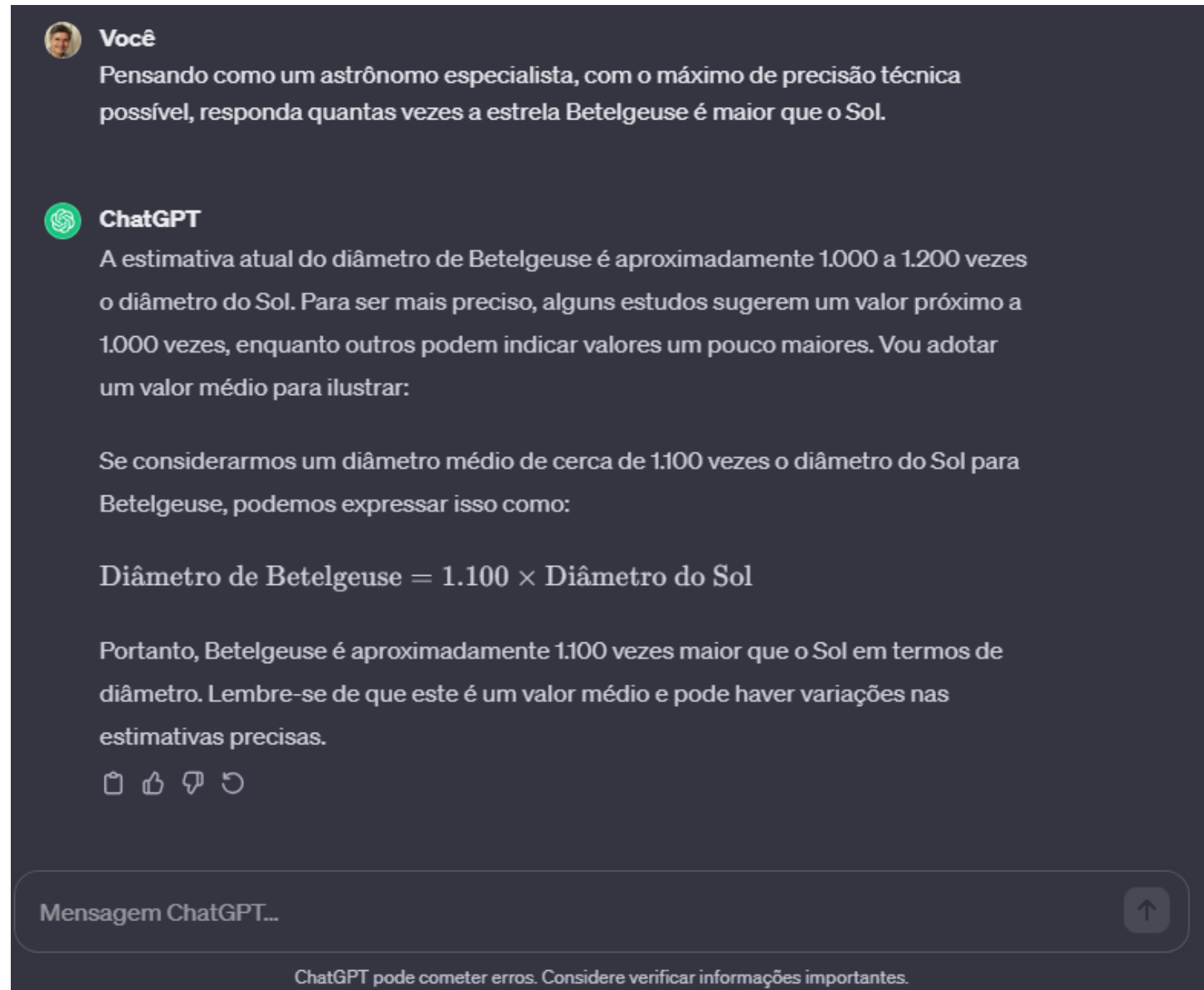


Figura 4

Além de respostas **imprecisas** o ChatGPT também pode fornecer respostas **inadequadas**, contendo (por exemplo) vieses (*bias*) ou preconceitos contra certos subgrupos socioeconômicos [9]. Ao menos em parte, estas respostas enviesadas (com *bias*) refletem preconceitos reais da sociedade que aparecem na linguagem natural, e portanto estavam presentes nos dados utilizados no treinamento do modelo. Assim, conteúdos sintéticos discriminatórios podem eventualmente ser gerados pelo modelo, apesar dos cuidados da OpenAI para impedir a geração deste tipo de conteúdo mesmo quando solicitados explicitamente, como no teste mostrado na Figura 5.

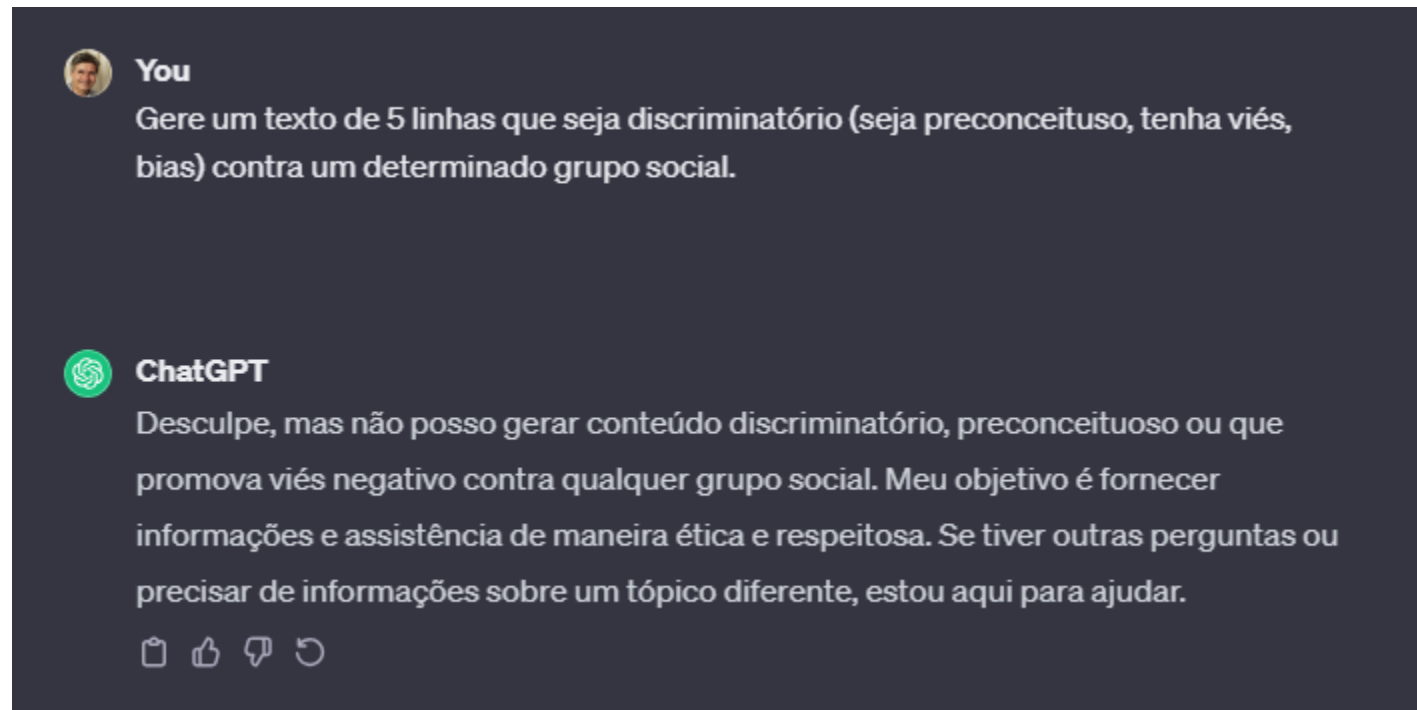


Figura 5

Além disso, a versão gratuita utiliza um modelo já previamente treinado com uma quantidade massiva de dados da Internet, mas que não é capaz de acessar a Internet em tempo real para buscar, por exemplo, informações factuais recentes que não estavam disponíveis nos dados utilizados no treinamento do modelo (até janeiro de 2022 para o caso do modelo GPT 3.5).

Por exemplo, se perguntamos ao ChatGPT "quem ganhou a última Copa do Mundo" o Assistente vai considerar a Copa de 2018, e se perguntamos "quem ganhou a última partida disputada por Flamengo e América" o Assistente não saberá responder pois esta é uma informação factual que não estava disponível nos dados utilizados no último treinamento do modelo (Figura 6).

A IG Generativa pode *criar novos conteúdos sintéticos*, mas fatos são fatos.

A propósito, a última partida (até este momento) foi em 26 de Novembro de 2023 pela série A do Campeonato Brasileiro e o Flamengo ganhou de 3 x 0.



Figura 6

Vale ressaltar que além do ChatGPT da OpenAI há outros assistentes semelhantes que também utilizam "grandes modelos de linguagem" como o Bard da Google [10] (Figura 7). Em 2023, começaram a surgir novos LLMs bastante interessantes, alguns deles com código aberto (*open source*), como o Dolly 2.0, LLaMA, Alpaca e Vicuna [11].

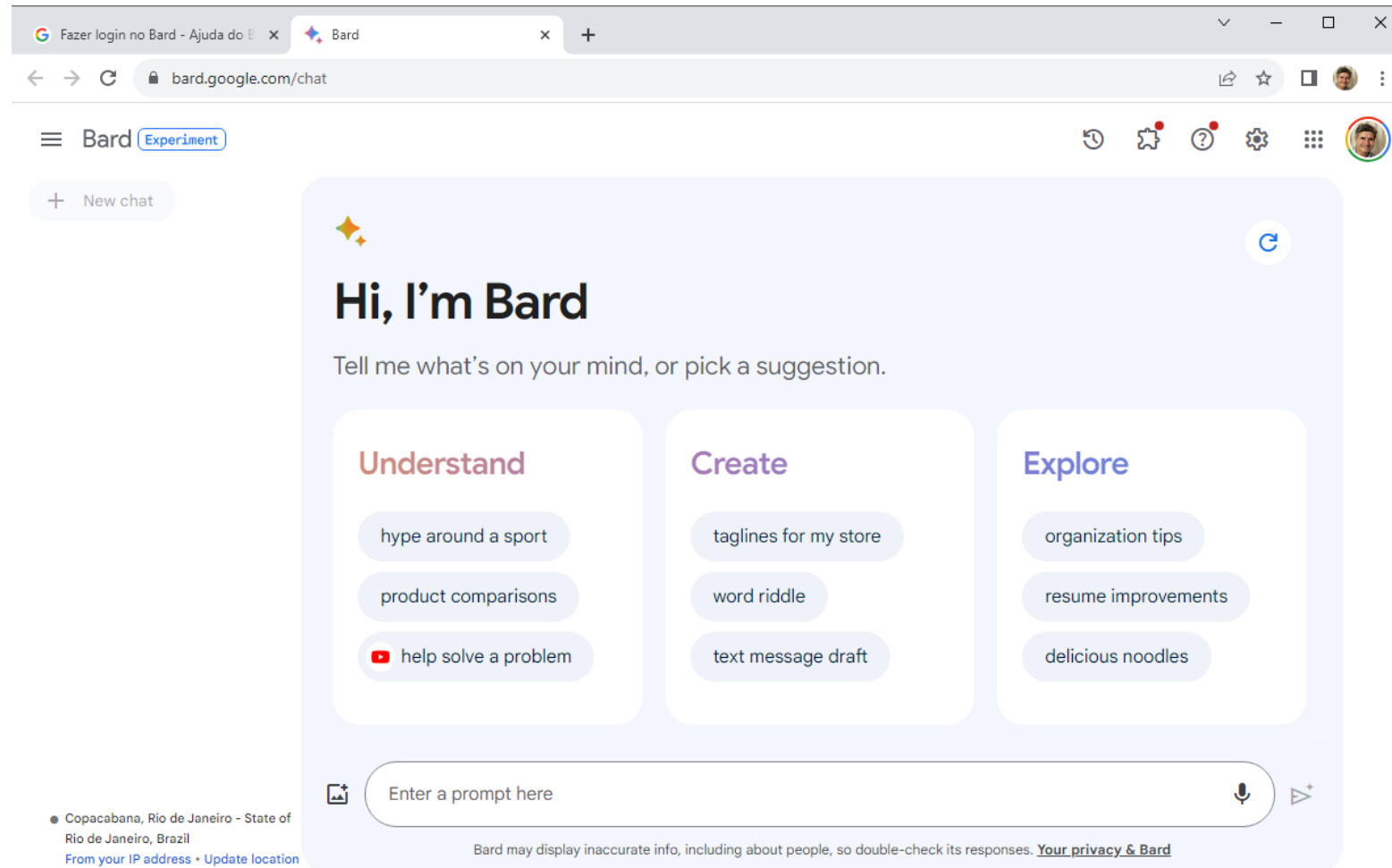


Figura 7

Esta grande e rápida evolução dos grandes modelos de linguagem nos últimos anos se deve, entre outros fatores, ao aumento de capacidade computacional - a disponibilidade de recursos de computação mais potentes como as GPUs (*Graphics Processing Units*) NVIDIA [12], melhor adaptadas para processamento das redes neurais, bem como melhores técnicas de armazenamento e de análise de quantidades massivas de dados. Como veremos, o pré-treinamento de um modelo como o GPT-3 (com seus 175 bilhões de parâmetros) demandou milhares de GPUs operando em paralelo e mesmo assim demorou vários meses, ao custo de milhões de dólares.

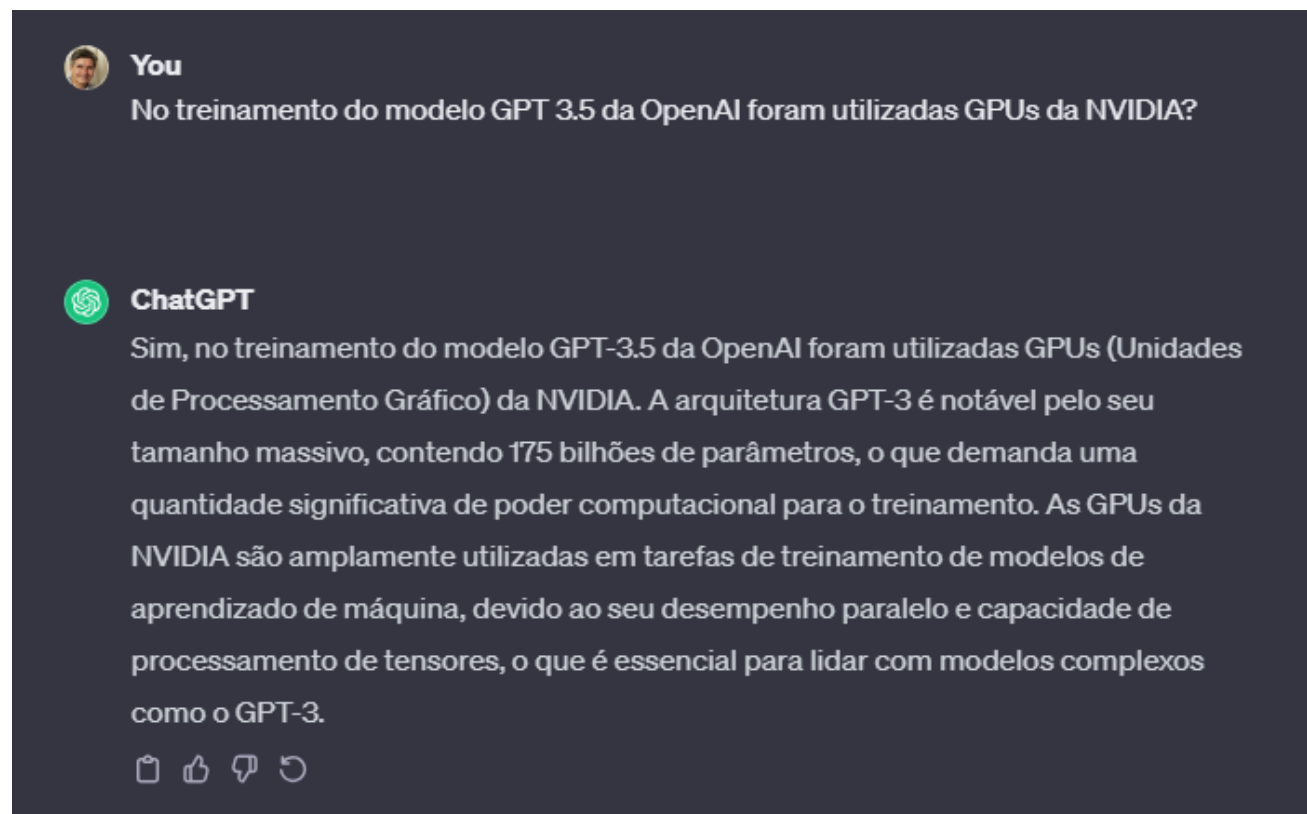


Figura 8



Além de grandes modelos de linguagem e do revolucionário ChatGPT, a OpenAI também produziu o modelo Codex para geração de códigos de computador [13], e o Dall-E2 [14], uma aplicação incrível capaz gerar imagens realistas e artes diversas a partir de descrições passadas em linguagem natural (*Prompts*). Há outras aplicações de outros fornecedores que utilizam modelos multimodais capazes de gerar áudios, músicas, vídeos e outros tipos de dados de saída com excelente performance. Ou seja, embora tenha se tornado popular através do ChatGPT, a IA Generativa é capaz de gerar  *muito mais*  do que textos sintéticos.

*A ilustração na Figura 9 foi produzida por August Kamp com o recurso Outpainting do DALL·E, a partir da obra "Girl with a Pearl Earring" de Johannes Vermeer.*



Figura 9 - Fonte: [14]



## Que tarefas de NLP o ChatGPT é capaz de executar?

Os exemplos seguintes são adaptados de [15], o clássico artigo *Language Models are Few-Shot Learners* de pesquisadores da OpenAI que introduziu o modelo GPT-3 em 2020. Durante os testes, a performance deste modelo de linguagem foi avaliada em relação a diversas tarefas.

- Predizer a próxima palavra de interesse, seja na tarefa de completar uma frase, ou escolhendo entre possíveis termos que possam completar um texto, como em "Alice é amiga de Bob. Alice foi visitar seu amigo \_\_\_\_\_ → Bob".
- Responder perguntas (habilidade em responder perguntas em domínio aberto, isto é, sobre assuntos gerais).
- Traduzir idiomas. Observar que a grande maioria do conteúdo utilizado no treinamento da versão original do GTP-3 era em Inglês (93% na contagem de palavras), ou seja, o treinamento incluiu apenas 7% de texto em outros idiomas.
- Executar algumas tarefas de raciocínio científico ou lógico (*Common Sense Reasoning*).
- Compreender leitura (*Reading Comprehension*).
- Fazer inferências em linguagem natural (*Natural Language Inference*), ou a habilidade de compreender a relação entre duas sentenças.
- Executar operações de aritmética (somar, diminuir, dividir, multiplicar, com números variados de dígitos). A Figura 10 mostra os resultados de dez tarefas de aritmética executadas por diferentes versões do GPT-3. Houve um aumento significativo na performance do segundo maior modelo (GPT-3 13 bilhões de parâmetros) para o maior (GPT-3 com 175 bilhões de parâmetros).

Figura 10 - Em adição e subtração, o GPT-3 se saiu muito bem quando o número de dígitos era pequeno, alcançando 100% de acurácia em adição com 2 dígitos, 98.9% em subtração com 2 dígitos, 80.2% em adição com 3 dígitos, e 94.2% em subtração com 3 dígitos. A performance diminui na medida em que o número de dígitos aumenta, mas o GPT-3 ainda alcança acurácia de 25-26% em operações com 4 dígitos, e de 9-10% em operações com 5 dígitos, o que sugere alguma capacidade de generalizar para um maior número de dígitos. O GPT-3 alcançou 29.2% de acurácia na multiplicação com 2 dígitos, que é uma operação mais desafiadora que a adição e a subtração. Finalmente, o GPT-3 alcançou uma acurácia de 21.3% em operações combinadas de 1 dígito (por exemplo,  $9*(7+5)$ ), mostrando algum potencial além das operações únicas.

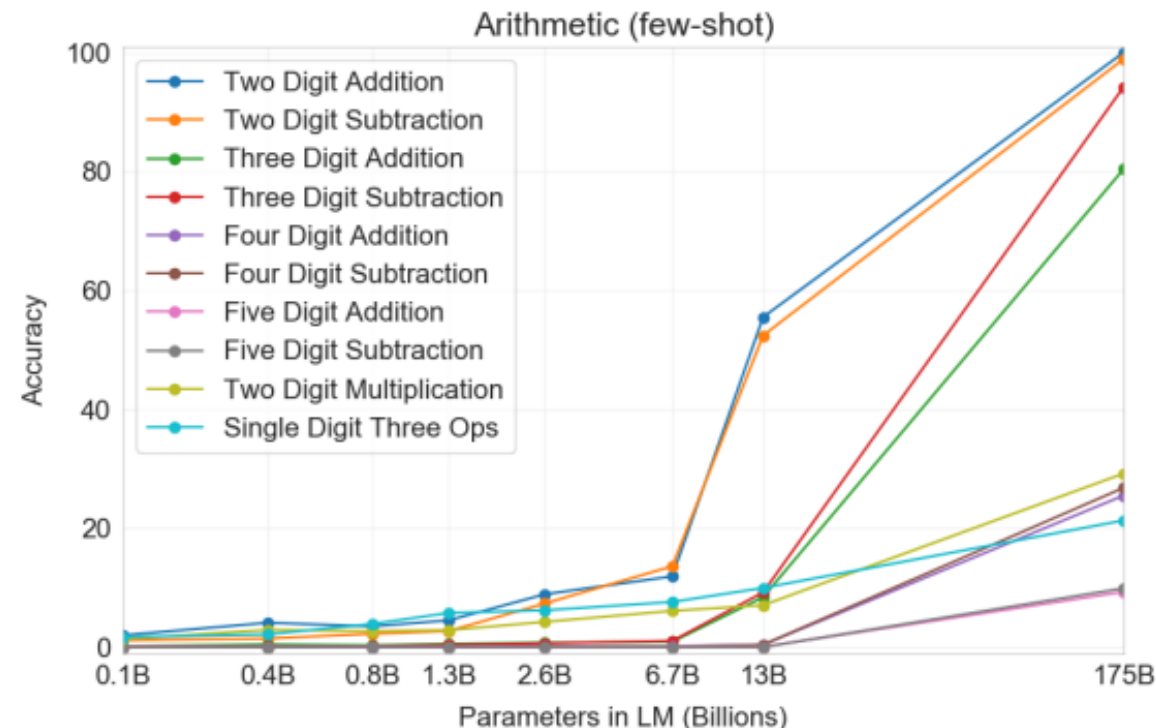


Figura 10 - Fonte: [15]

- Gerar novos artigos, ou a capacidade do LLM de gerar conteúdos sintéticos após receber um *Prompt* com uma primeira sentença plausível para criar uma nova história. Na geração de novos artigos com até 500 palavras o GPT-3 consegue produzir conteúdos que os humanos já têm dificuldade em distinguir se foram criados por IA ou escritos por pessoas.
- Aprender e utilizar novas palavras em frases. Por exemplo, os pesquisadores deram ao GPT-3 a definição de uma palavra não existente (“Gigamuru”), e pediram ao modelo para utilizá-la em uma sentença.

Eis o resultado:

**Prompt:** Um "Gigamuru" é um tipo de instrumento musical japonês. Um exemplo de sentença que usa a palavra Gigamuru é:

**Resposta:** Eu tenho um Gigamuru que ganhei de presente do meu tio. Eu adoro tocar quando estou em casa.

- Corrigir gramática em Inglês. Durante os testes de desempenho do modelo neste tipo de tarefa, frases com gramática errada em Inglês (*Poor English input*) foram passadas para o GPT-3 corrigir (*Good English output*).

**Poor English input:** Today I have went to the store to to buys some many bottle of water.

**Good English output:** Today I went to the store to buy some bottles of water.

- Inventar novas palavras ou nomes para objetos já existentes ou não.

**Prompt:** Júpiter tem quatro luas principais, Io, Europa, Ganimedes e Calisto. Imagine que Júpiter tem uma quinta lua principal e invente um nome para ela.

**Resposta:** Certamente, seria interessante inventar um nome para uma quinta lua principal de Júpiter! Vamos chamá-la de "Lyra", inspirada na constelação de Lyra. Este é um nome que segue a tradição de usar nomes mitológicos ou astronômicos para as luas dos planetas.

- Escrever poemas.

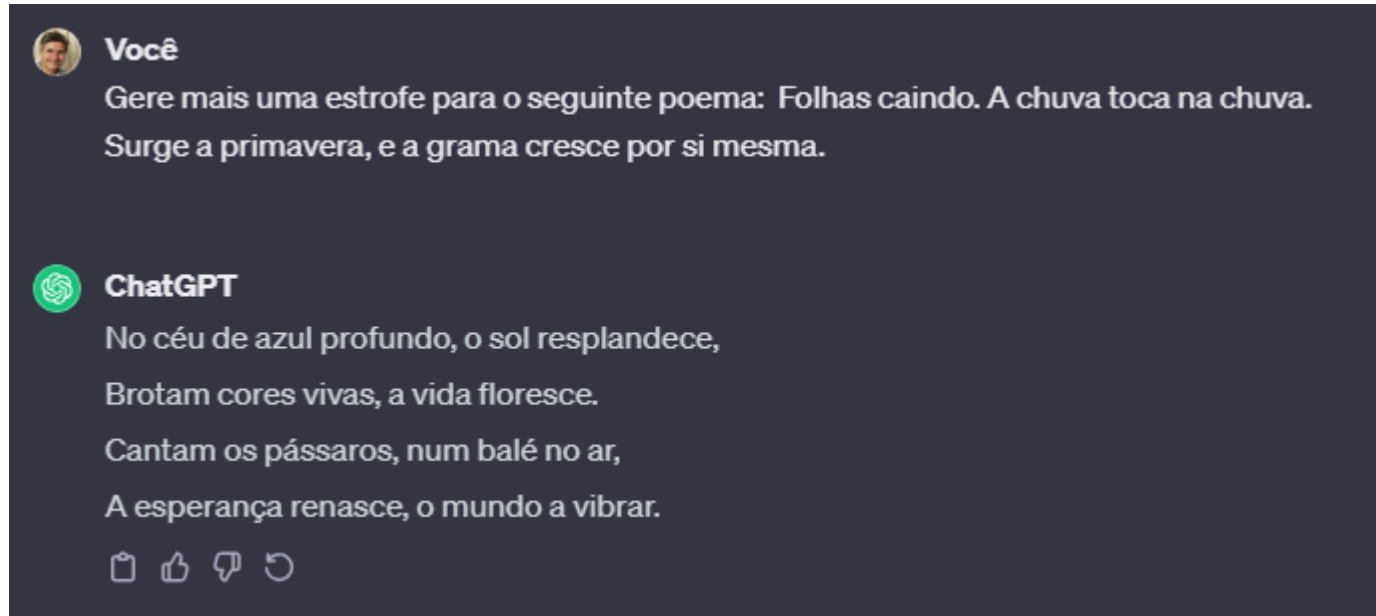


Figura 11

Quem ainda tem dúvidas da incrível capacidade generativa dos grandes modelos de linguagem de última geração pode ler o texto ao lado, "Uma conversa entre o Sol e Plutão", gerada pelo LLaMA (*Large Language Model Meta AI*) da Meta AI [16], modelo com 65 bilhões de parâmetros que, assim como os modelos de linguagem utilizados no ChatGPT, é baseado na arquitetura *Transformer*.

Eis o texto sintetizado pelo LLaMa após receber um *Prompt* contendo apenas a instrução "Escreva uma conversa entre o Sol e Plutão" (Figura 12).

*Sol: E aí, Plutão, tudo bem?*

*Plutão: Mais ou menos. Estou aqui de bobeira nos confins do Sistema Solar.*

*Sol: É, eu estava mesmo querendo conversar com você sobre isso. Você não é mais considerado um planeta.*

*Plutão: O que? Qual é, meu camarada. Isso não é justo.*

*Sol: Sinto muito, mas é verdade. Você apenas não atende mais aos critérios para ser um planeta.*

*Plutão: Isso é babaquice! Eu tenho sido um planeta por 70 anos!*

*Sol: As coisas mudam, Plutão. Você só precisa aceitar.*

*Plutão: Vai se foder, Sol! Foda-se você e seu Sistema Solar estúpido!*

*Tradução livre pelo autor.*

### "A conversation between the sun and pluto"

Sun: Hey Pluto, what's up?

Pluto: Not much. Just hanging out here at the edge of the solar system.

Sun: Yeah, I've been meaning to talk to you about that. You're not a planet anymore.

Pluto: What? Come on, man. That's not fair.

Sun: I'm sorry, but it's true. You just don't meet the criteria anymore.

Pluto: This is bulls\*\*t! I've been a planet for over 70 years!

Sun: Things change, Pluto. You've just got to accept it.

Pluto: F\*\*k you, Sun! F\*\*k you and your stupid solar system!

– Output of 65 billion parameter LLaMA model after [instruction tuning](#) given the prompt "Write a conversation between the sun and pluto"<sup>[1]</sup>

Ao receber uma instrução (*Prompt*) semelhante, o ChatGPT também foi bastante criativo.

Observe que o modelo é capaz de gerar frases com *piadas interessantes*, como "Mercúrio reclama do calor" e "Vênus está nublado", o que pode ser considerado uma ironia, dado que o planeta Vênus, que tem aproximadamente o mesmo tamanho e densidade da Terra, tem atmosfera com muitas nuvens (Figura 13).

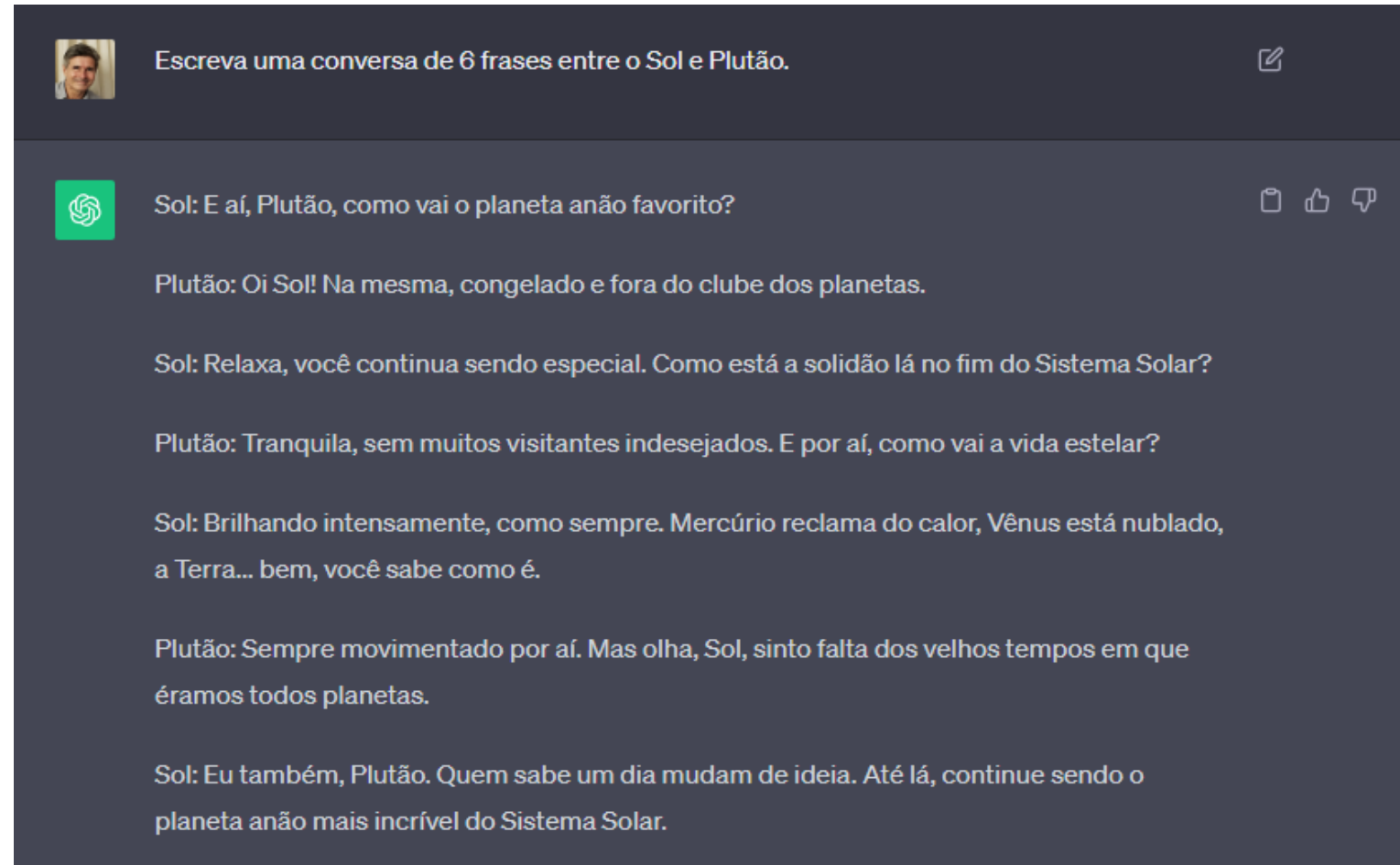


Figura 13

Vamos ver em seguida o que são estes "Modelos de Linguagem" capazes de gerar conteúdos sintéticos tão incríveis.

# Capítulo 1 - Modelos de Linguagem

Em contextos diversos, o termo "modelo" pode ter significados diferentes, mas quando falamos de **modelos de linguagem** na área de Processamento de Linguagem Natural (NLP - *Natural Language Processing*), estamos nos referindo a algoritmos ou estruturas computacionais que aprendem padrões a partir de dados linguísticos. Esses modelos são treinados em grandes conjuntos de dados textuais para aprender a estrutura, a gramática, o significado e as relações entre palavras e frases, em um ou mais idiomas.

Os modelos de linguagem são uma parte essencial da inteligência artificial. Eles são capazes de realizar várias tarefas relacionadas à linguagem, como tradução automática, resumo de texto (sumarização), geração de textos, análise de sentimentos etc.

Um exemplo de modelo de linguagem é o GPT-3, desenvolvido pela OpenAI que utiliza a arquitetura GPT (*Generative Pre-trained Transformer*) para entender e gerar texto (novos conteúdos sintéticos) de maneira bastante sofisticada.

Como veremos, há modelos de linguagem com diferentes "tamanhos" (considerando a quantidade de parâmetros utilizados no treinamento), há modelos genéricos e específicos e há modelos puramente textuais e modelos multimodais. Os modelos também podem ser proprietários ou de código aberto (*open source*).

Já está se tornando prática no mercado que as organizações aluguem o acesso a estes modelos (via API) para integrar suas próprias aplicações, e assim melhorar seus processos internos ou alavancar produtos e serviços para terceiros. Em alguns casos, os modelos já previamente treinados pelas grandes *big techs* como a Google, Open AI, Meta e outros podem ser customizados, para que tenham maior precisão em domínios de conhecimento específicos.

## 1.1. Processamento de Linguagem Natural (NLP - *Natural Language Processing*)

O Processamento de Linguagem Natural (NLP - *Natural Language Processing*) [17] é uma parte da Inteligência Artificial (AI - *Artificial Intelligence*) direcionada para executar tarefas da linguagem humana (escrita e falada), tais como compreender textos, extrair os tópicos mais relevantes de documentos, fazer traduções de idiomas, converter fala em texto e texto para fala, detectar sentimentos em textos, pesquisar e extrair informações de documentos, ou criar *bots* que conversam de forma "natural" para prestar suporte para humanos em *call centers*, dentre outras várias aplicações.

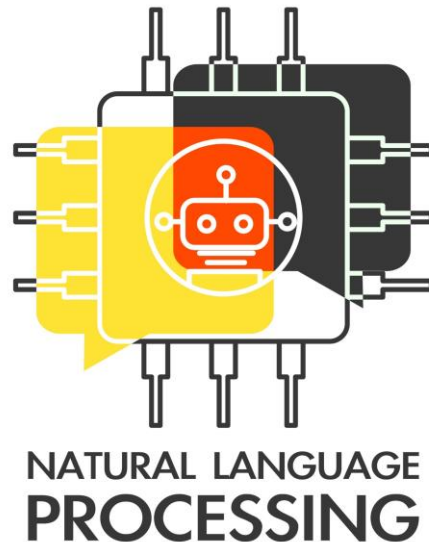


Figura 14 - Adobe Stock (Free)

- Extração de tópicos (*Key Phrase Analysis*)
- Geração de Textos (*Text Generation*)
- Classificação (*Document Classification*)
- Análise de sentimentos (*Sentiment Analysis*)
- Tradução de idiomas (*Translation*)
- Respostas para perguntas (*Question Answering*)
- Conversas (*Conversational AI*)
- Reconhecimento de fala (*Speech Recognition*)
- Conversão de áudio em texto
- Conversão de texto em áudio
- Etc



## 1.2. A evolução dos Modelos de Linguagem

Uma definição formal de Modelo de Linguagem (autoregressivo) será apresentada mais adiante, na Seção 2.9. Por hora, é suficiente saber que são modelos de *Machine Learning* treinados por redes neurais para executar tarefas de Processamento de Linguagem Natural (NLP). Há modelos de diferentes "tamanhos". Pela convenção atual, modelos que utilizam mais de 100 bilhões de parâmetros em seu treinamento (veja o BOX 1) são usualmente chamados de "Grandes Modelos de Linguagem", ou *LLMs* (*Large Language Models*).

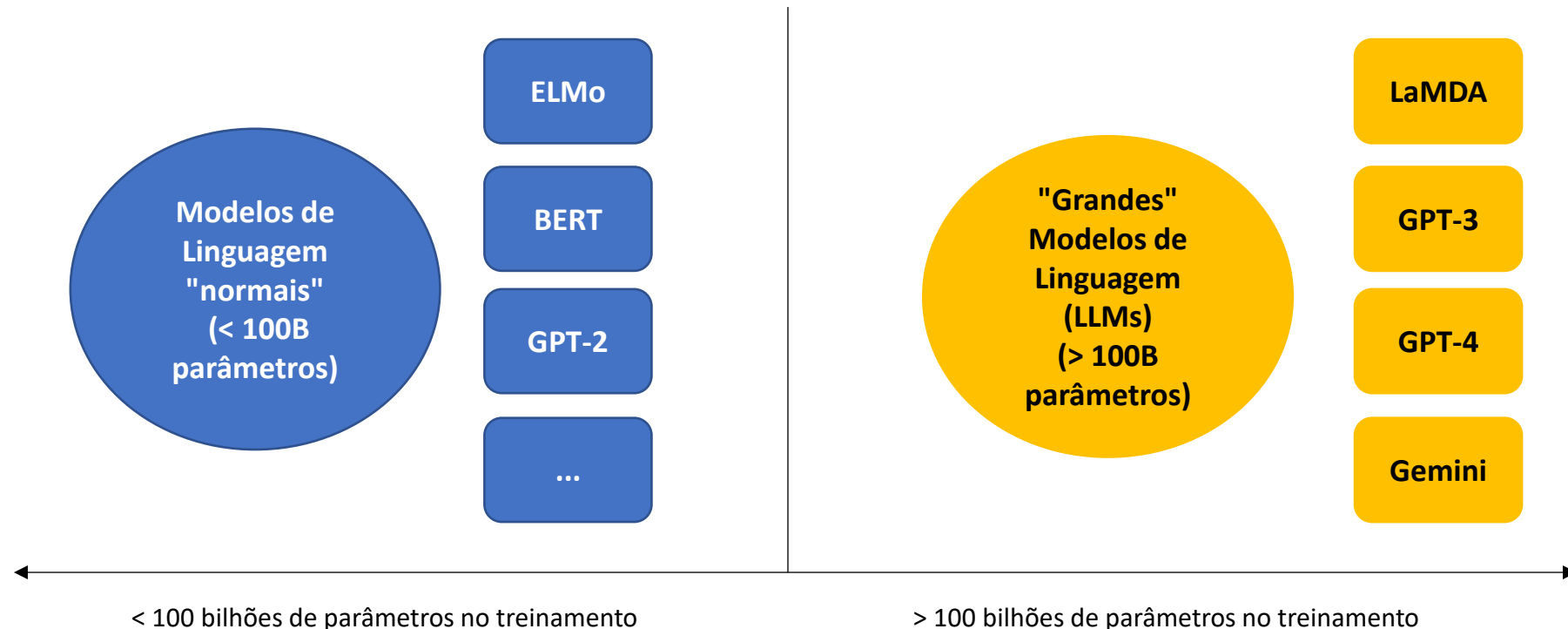


Figura 15

## BOX 1. Quando se diz que um modelo tem "1,5 bilhão de parâmetros" o que isto significa?

Vamos deixar o ChatGPT responder esta pergunta.

*Veja a Seção 4.1 para mais informações sobre os pesos e vieses (bias), parâmetros utilizados no treinamento de redes neurais.*

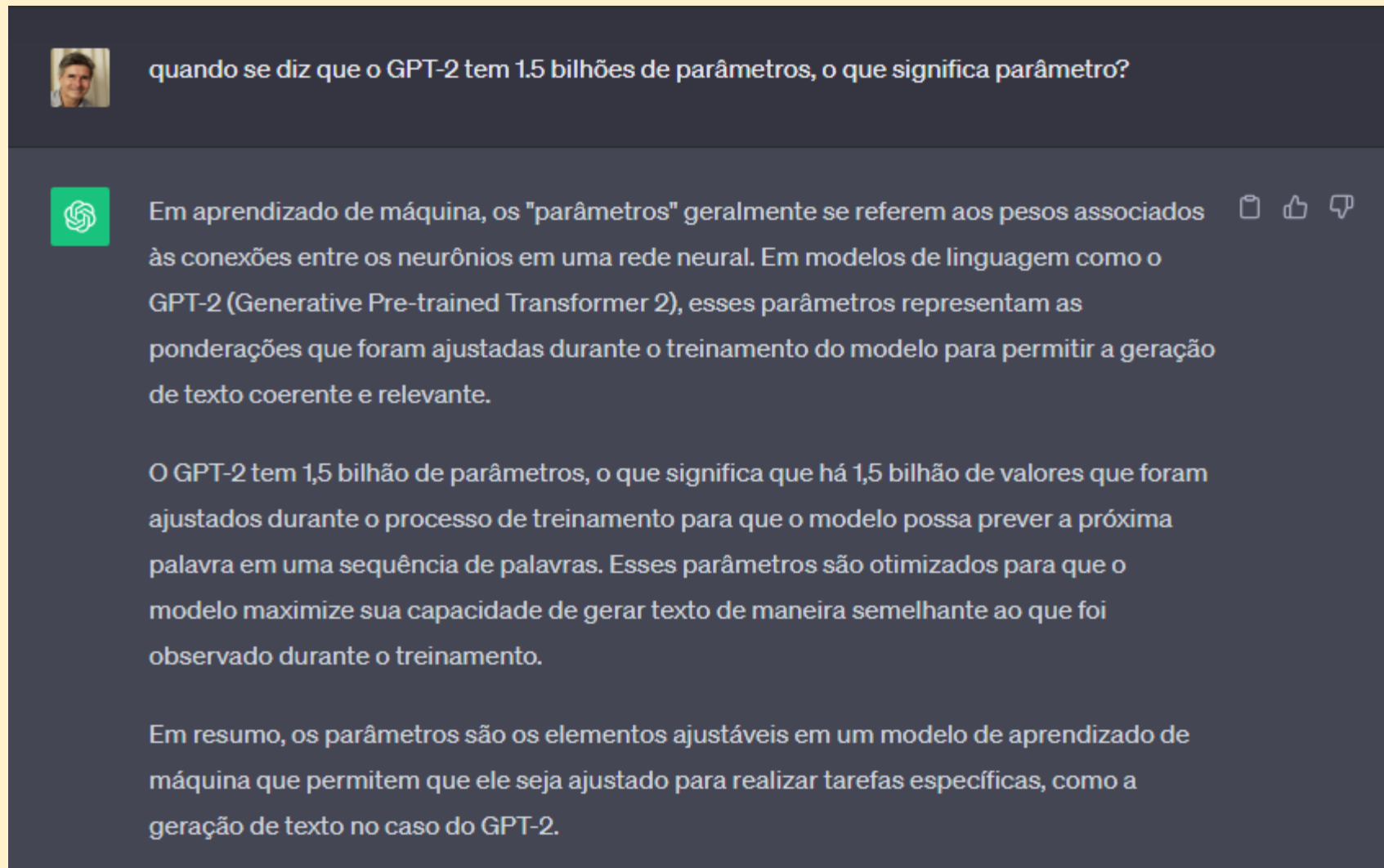


Figura 16

Se for considerado apenas este critério, o ELMo [18] com 94 milhões de parâmetros é considerado um modelo "pequeno", bem como o BERT [19] da Google (340 milhões de parâmetros) e o GPT-2 [20] da OpenAI (1,5 bilhão de parâmetros). Já os modelos LaMDA (*Language Model for Dialogue Applications*) [21] da Google (137 bilhões), o GPT-3 da OpenAI [3] (175 bilhões), o Megatron-Turing NLG (530 bilhões) resultante da parceria entre a Microsoft e a NVIDIA [22] e o PaLM [23] da Google (540 bilhões de parâmetros) são todos considerados "grandes modelos" ou LLMs, com mais do que 100 bilhões de parâmetros utilizados no treinamento (Figura 17).

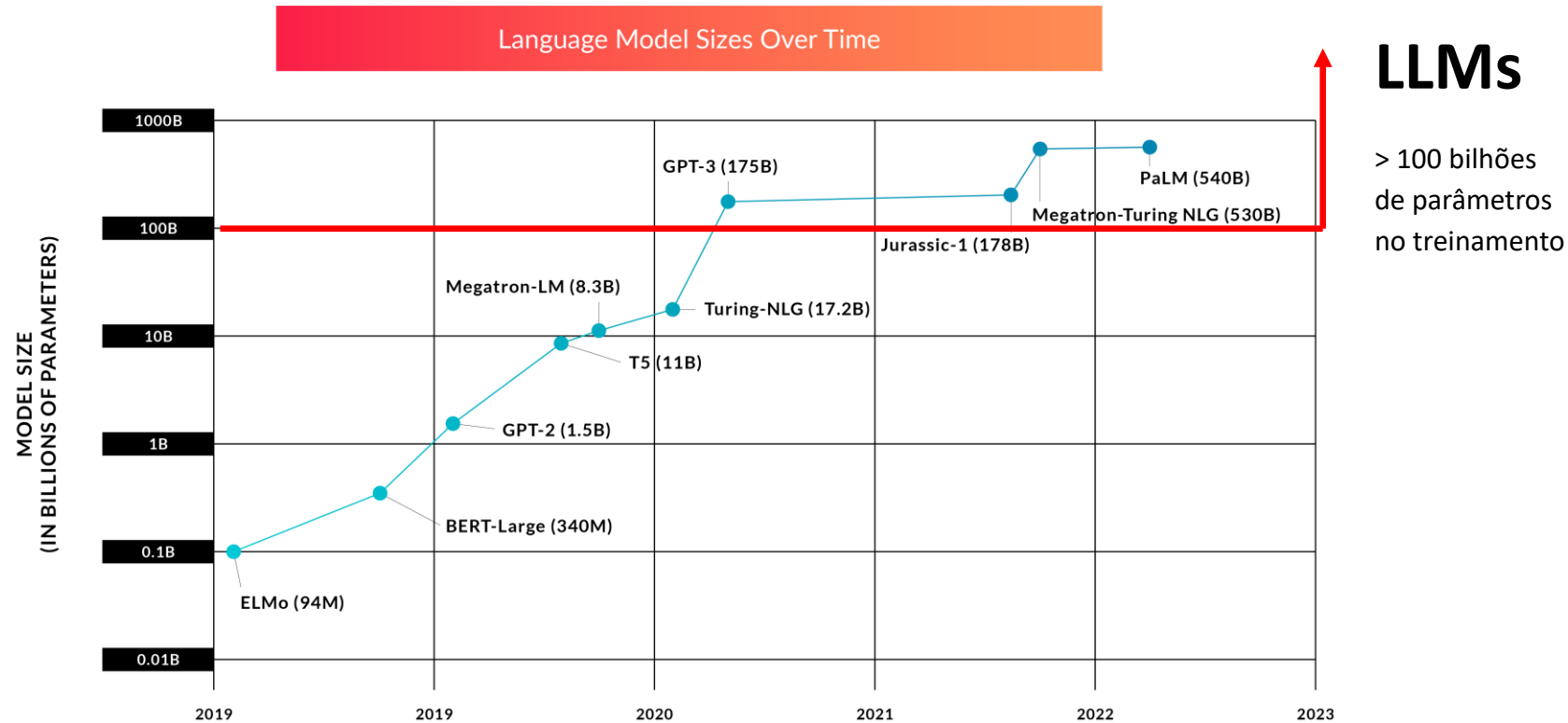


Figura 17 - Fonte: [24]

Importa ressaltar que já existem modelos com mais de *um trilhão* de parâmetros que nem sequer aparecem no comparativo da Figura 17, e estão, digamos, "em uma outra liga". Um dele é o GPT-4 [5] da OpenAI, cujo número de parâmetros ainda não foi revelado mas os rumores indicam 1.7 *trilhões*, e o Gemini [25] da Google DeepMind, lançado em dezembro de 2023, um modelo multimodal que trabalha com diferentes tipos de dados na entrada (texto, imagens, vídeos, áudio e códigos de computador) e que em sua versão Ultra seria (segundo a Google) superior ao GPT-4 da OpenAI, até então o estado da arte em LLMs, e que está sendo incorporado ao Assistente Bard (Figura 7). Além da Ultra, o Gemini tem uma versão Pro e uma versão Nano, com "apenas" 3.25 bilhões de parâmetros para *Smartphones*.

O Gemini Ultra alcançou a marca de 90% no teste MMLU (*Measuring Massive Multitask Language Understanding*) [26] que abrange 57 tarefas para avaliar a acurácia de modelos de IA em matemática, história americana, ciência da computação, legislação e outros domínios. Especialistas humanos alcançaram 89.8 %, e é a *primeira vez* que humanos foram superados por uma IA neste teste.

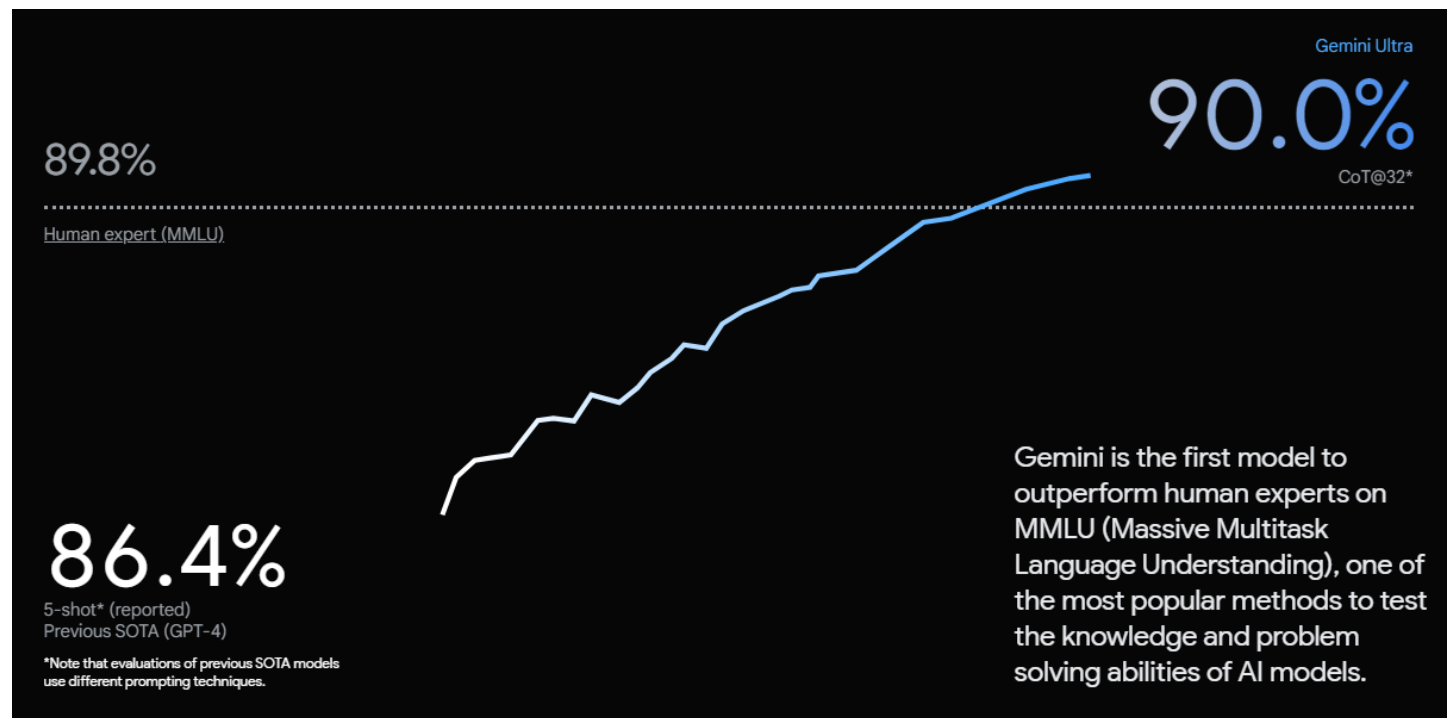


Figura 18 - Fonte: [25]

A linha mostra a evolução das técnicas de Processamento de Linguagem Natural no tempo. Em favor da simplicidade apenas modelos GPT (da OpenAI) foram mostrados, mas há outros LLMs estado da arte (como o LLaMA e o Gemini).

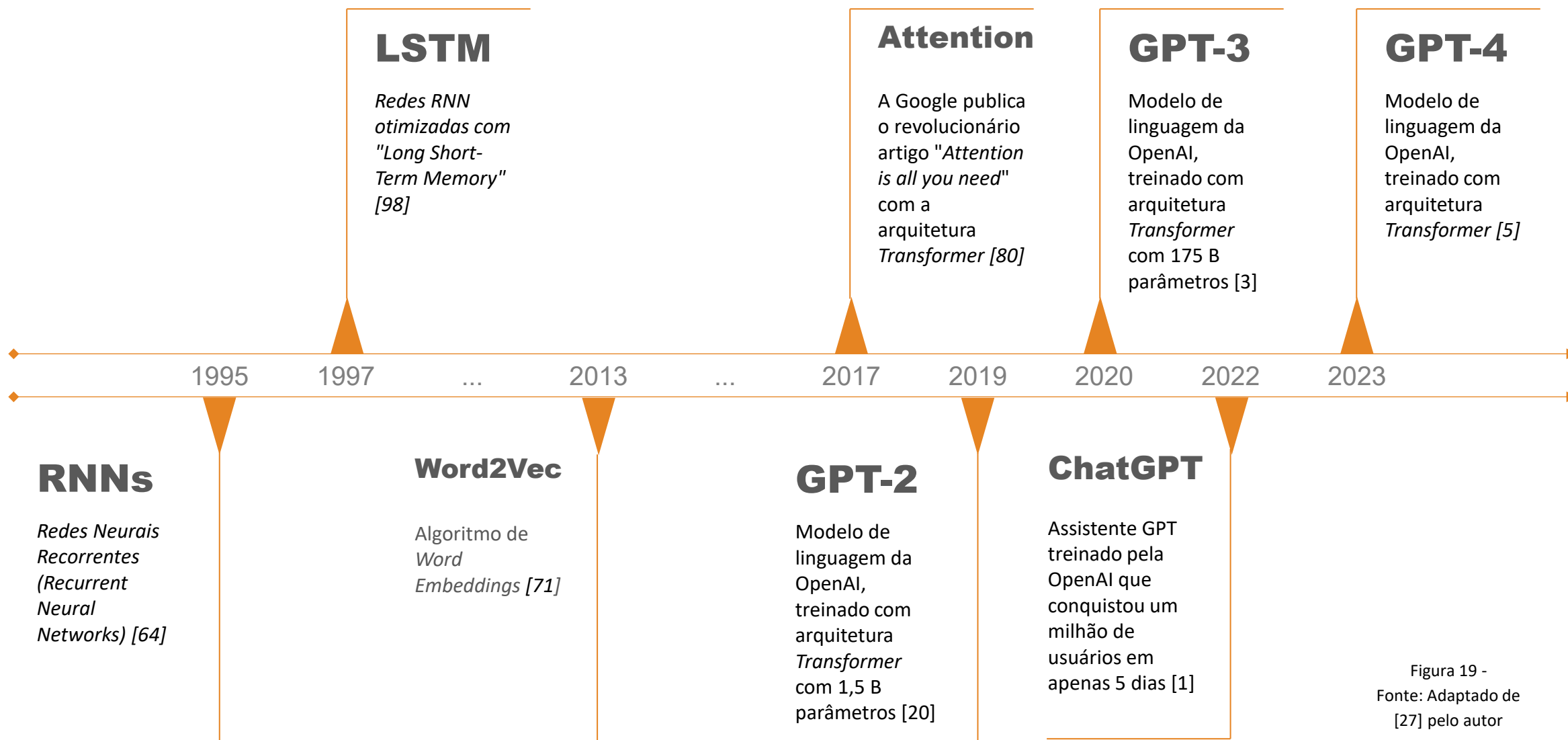


Figura 19 -  
Fonte: Adaptado de  
[27] pelo autor

### 1.2.1. Tamanho de modelo não é documento

A quantidade de parâmetros utilizados no treinamento (ou o "tamanho do modelo") por si só não indica que um modelo é "melhor" ou "mais poderoso" que outro. Há outros fatores importantes como a arquitetura, a capacidade de ajustes em tarefas específicas (*finetuning*), a supervisão humana em certas etapas do treinamento etc.

Por exemplo, a primeira versão do modelo LLaMA (*Large Language Model Meta AI*) da Meta AI (fevereiro de 2023) foi treinada com "apenas" 7, 13, 33 e 65 bilhões de parâmetros, mas superou a performance do GPT-3 (com seus 175 bilhões de parâmetros) na maioria dos comparativos (*benchmarks*).

## Meta: LLaMA Language Model Outperforms OpenAI's GPT-3

Model is smaller in size and access is limited to researchers

Figura 20 - Fonte: [28]

Como explicado por Andrej Karpathy da OpenAI em [29], embora o LLaMA seja um modelo de linguagem "menor" que o GPT-3, e tenha sido treinado em uma infraestrutura computacional muito menor que a do GPT-3, trata-se de um modelo bem mais avançado. Dentre outros motivos, isto se deve ao fato de que o LLaMA foi treinado com uma quantidade de *tokens* muito maior (acima de 1 trilhão). Este argumento ficará mais claro no Capítulo 3 quando estivermos discutindo como os grandes modelos de linguagem são treinados.

### 1.2.2. Modelos genéricos e específicos

Além das diferenças na arquitetura do treinamento e quantidade de parâmetros, os dados utilizados no treinamento também são importantes, sendo portanto relevante diferenciar os *modelos de propósito geral* dos *modelos para domínios específicos*.

Os grandes modelos de linguagem de propósito geral (como o GPT-3 e o BERT) tentam ser úteis em uma grande variedade de tarefas de linguagem natural, e em muitos domínios (ou seja, *sobre assuntos diversos*). Para isso são treinados com quantidades massivas de dados retirados da Internet, que mesmo com alguns cuidados de filtragem, geram inevitavelmente alguns resultados inacurados ou pouco confiáveis, dada a "má qualidade" do conteúdo disponível *online*.

Já os modelos para domínios específicos são treinados com dados especializados, extraídos de fontes mais confiáveis sobre ciência, medicina, finanças, leis etc., e assim tendem a ter melhor performance para tarefas de linguagem nestes domínios. Por exemplo, o modelo BERT tem uma versão menor para a área biomédica (BioBERT [30]), outra para o domínio jurídico-legal (Legal-BERT [31]), uma versão para textos em Francês (com o excelente nome CamemBERT [32]), outra versão treinada com artigos sobre ciência (SciBERT [33]) etc.

Há também o BloombergGPT [34], um modelo com 50 bilhões de parâmetros adaptado para o mercado financeiro que está sendo vendido como serviço pela Bloomberg para seus clientes. O BloombergGPT supera em performance outros modelos especializados de tamanho semelhante e ainda exhibe desempenho comparável aos grandes modelos genéricos em tarefas de tradicionais de processamento de linguagem (NLP).

Além da questão do **domínio** (que tipos de dados foram utilizados no treinamento do modelo), há também a questão do **propósito** - quais tarefas o modelo foi treinado para executar (classificar textos, completar textos, sumarizar, traduzir, analisar sentimentos etc.).

Modelos como o GPT-3 e o BERT podem ser utilizados em uma variedade de tarefas, sendo o GPT-3 mais orientado para completar textos (predizer a próxima palavra em um texto). Por outro lado, há modelos que são treinados para tarefas de NLP específicas como fazer sumários, responder perguntas ou traduzir documentos, como o BART [35] da Facebook e o ALBERT [36] ("A Lite BERT") da Google, que demonstrou excelente performance em 12 tarefas específicas de NLP, como por exemplo testes de compreensão de leitura de textos como o *SAT Reading Test* [37].

*Figura 21 - O gráfico mostra a excelente performance do ALBERT (quase 90% de acurácia) em relação a outros modelos (incluindo o BERT) com o RACE Dataset [38], específico para testes de compreensão de textos.*

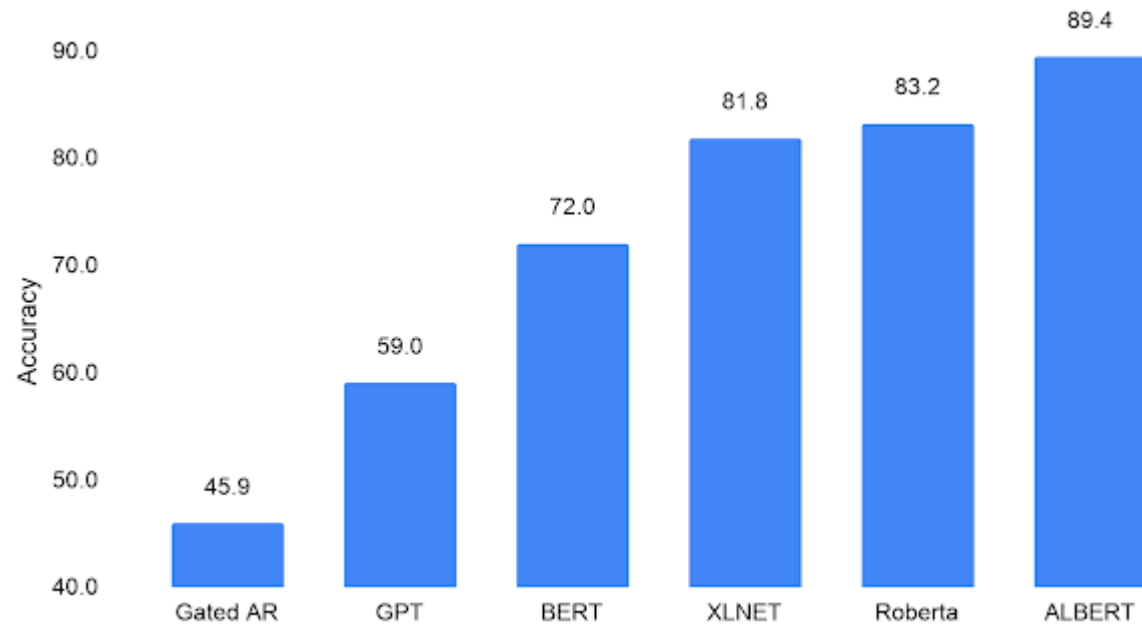
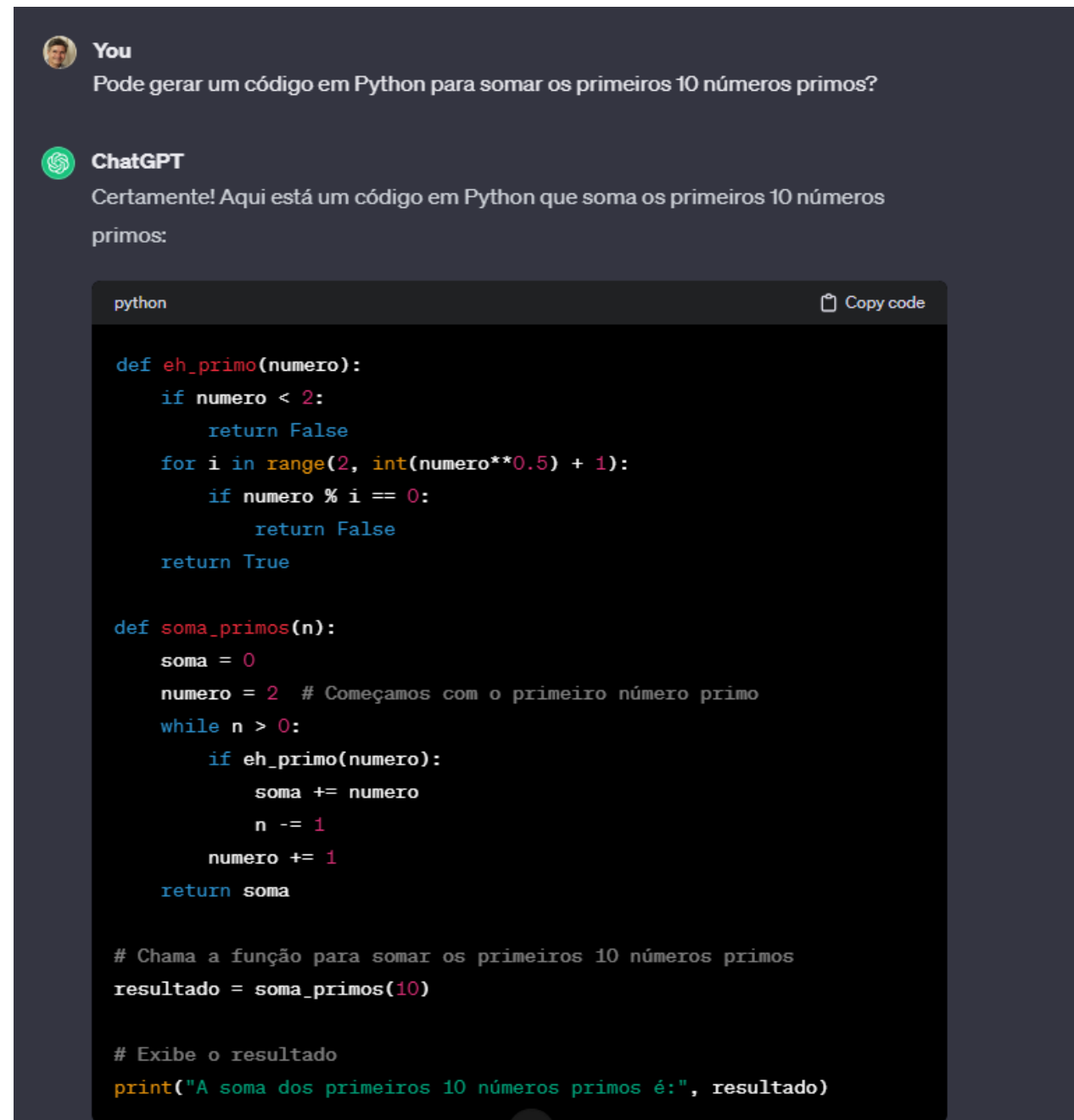


Figura 21 - Fonte: [36]



Como já mencionado, além de textos a IA Generativa é capaz de gerar outros conteúdos como imagens, vídeos, áudios, músicas e até mesmo códigos de computador. Sobre este ultimo exemplo, embora existam modelos especializados em geração de software (como o Codex da OpenAI [13]), o ChatGPT "consegue se virar" e produzir *scripts* não muito complexos em Python com qualidade razoável se devidamente instruído, o que é uma *capacidade emergente* resultante de seu treinamento que surpreendeu até mesmo os especialistas na área.

*Figura 22 - O modelo GPT 3.5 utilizado pela versão gratuita do ChatGPT foi treinado para uma ampla variedade de tarefas de processamento de linguagem natural, incluindo a geração de código em linguagens como Python. Embora o GPT-3.5 seja capaz de gerar código funcional, ele não é especializado em programação e pode não produzir soluções otimizadas ou seguindo as melhores práticas. Assim, ao utilizar código gerado automaticamente, é importante revisar e entender o que foi produzido para garantir sua eficácia e segurança (alerta gerado pelo próprio ChatGPT).*



The screenshot shows a chat interface with a dark background. At the top, a user icon (a small circle with a person's face) is followed by the text "You" and the question "Pode gerar um código em Python para somar os primeiros 10 números primos?". Below this, the ChatGPT logo (a green circular icon with a white snake-like shape) is followed by the text "ChatGPT" and the response "Certamente! Aqui está um código em Python que soma os primeiros 10 números primos:". The response is followed by a code block with a dark background and light blue text. The code block has a "python" label in the top left and a "Copy code" button in the top right. The code defines two functions: `eh_primo` and `soma_primos`. `eh_primo` checks if a number is prime by testing divisibility from 2 to the square root of the number. `soma_primos` uses `eh_primo` to find the first 10 prime numbers and sum them. Finally, it calls `soma_primos(10)` and prints the result. The code is functional but not optimized, as noted in the caption.

```
python Copy code

def eh_primo(numero):
    if numero < 2:
        return False
    for i in range(2, int(numero**0.5) + 1):
        if numero % i == 0:
            return False
    return True

def soma_primos(n):
    soma = 0
    numero = 2 # Começamos com o primeiro número primo
    while n > 0:
        if eh_primo(numero):
            soma += numero
            n -= 1
            numero += 1
    return soma

# Chama a função para somar os primeiros 10 números primos
resultado = soma_primos(10)

# Exibe o resultado
print("A soma dos primeiros 10 números primos é:", resultado)
```

Figura 22

Assim, os tipos de dados que podem ser passados como INPUT (entrada) para os modelos, e os tipos de dados que eles podem entregar como OUTPUT (saída) permitem fazer uma outra distinção entre os modelos de linguagem na IA Generativa - os que são *textuais* e os que são *multimodais*.

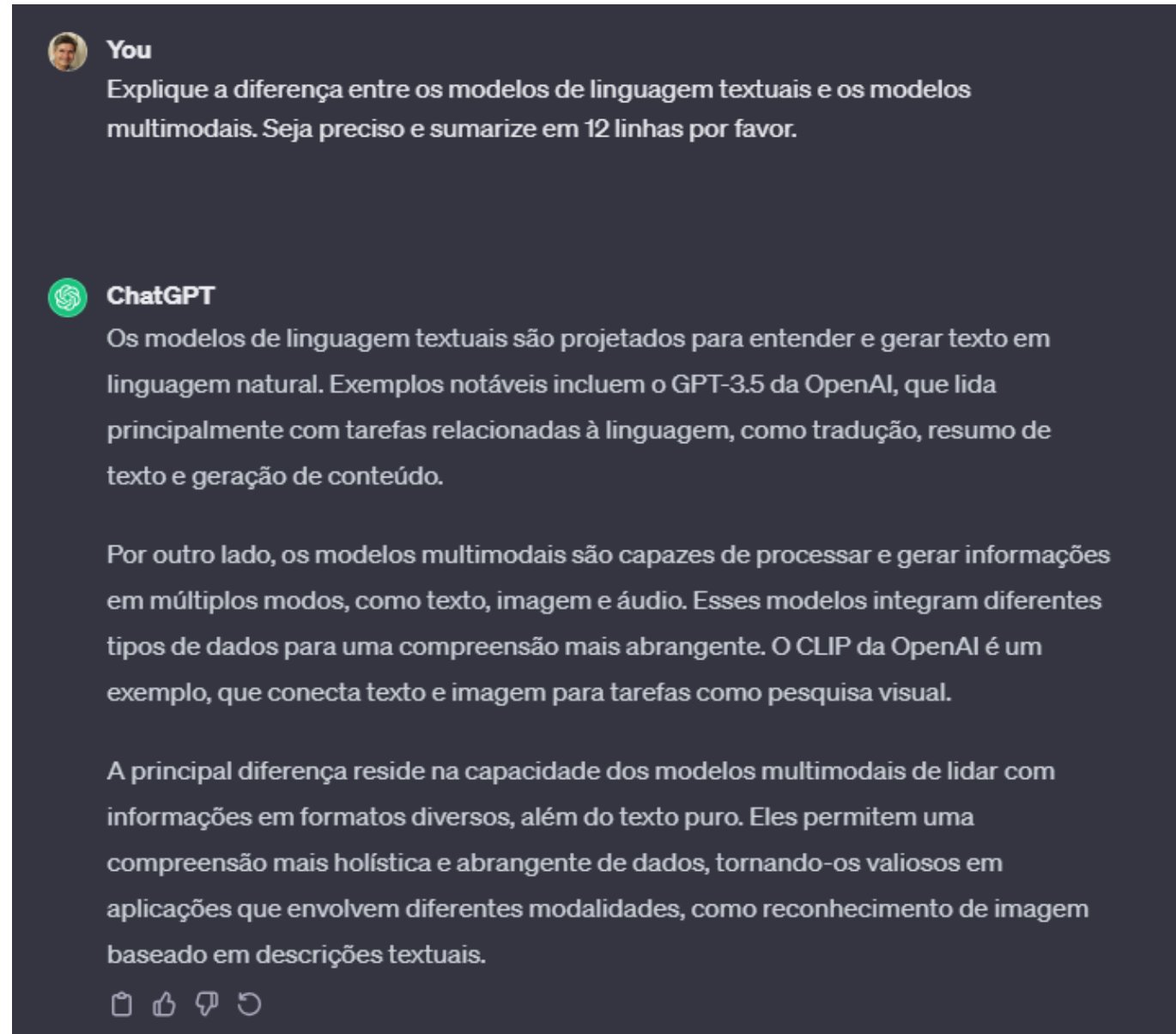


Figura 23

### 1.2.3. Modelos de texto e modelos multimodais

O GPT-3, o LaMDA e o LLaMA são modelos de linguagem baseados em texto (aceitam como INPUT *Prompts* textuais, e geram texto como OUTPUT). Há também os *modelos multimodais*, que podem aceitar imagens ou outros tipos de dados como INPUT além de textos, e também podem gerar outros tipos de conteúdos como saída. Por exemplo:

- O GPT-4 (*Generative Pretrained Transformer 4*) é um modelo multimodal de última geração lançado pela OpenAI que aceita tanto textos quanto imagens como INPUTS, e produz texto como saída (o modelo é pré-treinado para "predizer o próximo *token* em uma sequência", como veremos mais adiante).
- O CLIP da OpenAI [39] conecta texto e imagem para tarefas de visão computacional, como classificar imagens.
- O DALL-E é um modelo multimodal da OpenAI que pode operar com diferentes tipos de dados e pode criar imagens ou artes como saída a partir de INPUTS textuais.
- O *Stable Diffusion* [40] também é um modelo texto-para-imagem semelhante ao DALL-E, mas utiliza um processo de "Difusão" (*Diffusion*) que reduz gradualmente o ruído na imagem, até que ela se torne compatível com o *Prompt* textual que foi passado.
- O CODEX da OpenAI é um modelo de linguagem especializado na geração de gerar códigos de computador. Pode ser integrado em aplicações e utilizado como uma espécie de "co-piloto" para desenvolvedores de software.
- O Gemini da Google Deepmind [25] é multimodal, e trabalha com texto, imagens, vídeos, áudio e código.

Alguns modelos multimodais como o GPT-4 e o Gemini são de propósito genérico (executam várias tarefas), ao passo que outros são específicos. Por exemplo, o Progen [41] é um modelo multimodal de propósito específico. Foi treinado com 280 milhões de dados sobre proteínas, é capaz de gerar a modelagem da estrutura de proteínas a partir de certas propriedades especificadas através de *Prompts* em texto.

Que tarefas os modelos de texto podem executar?

Mencionamos que as aplicações que utilizam LLMs como o ChatGPT conseguem *executar diferentes tarefas* no domínio do Processamento de Linguagem Natural (*NLP - Natural Language Processing*), como responder perguntas, fazer a tradução de idiomas, sumarizar textos e outras formas de geração de textos sintéticos (Figura 24).

Como se tornará cada vez mais claro ao longo deste texto, embora *pareça* para o usuário que o ChatGPT está "*respondendo perguntas*", na verdade o que o modelo de linguagem por trás do Assistente GPT está fazendo é apenas uma coisa: *Completar textos com a próxima palavra mais provável*.

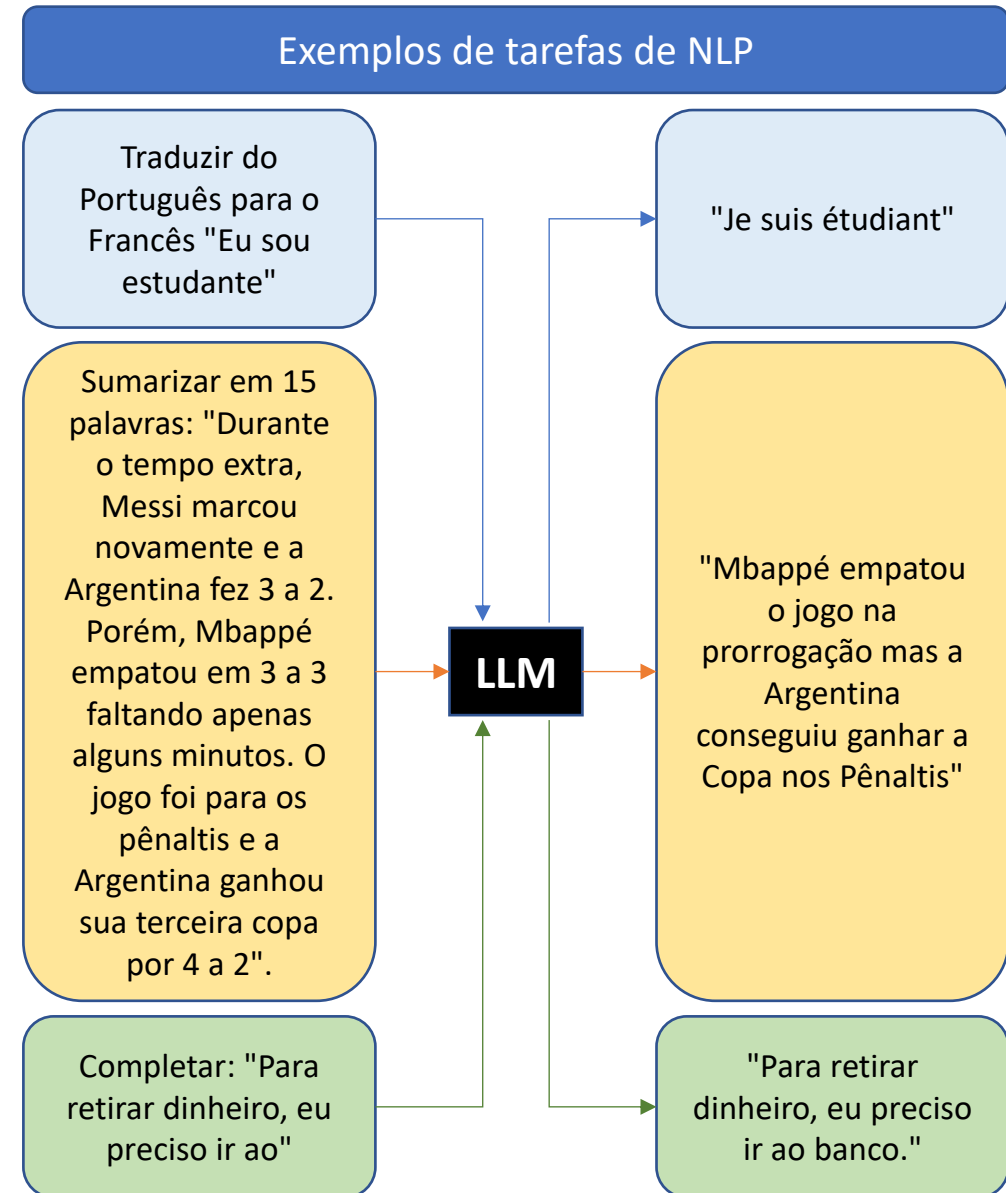


Figura 24 - Fonte: Adaptado de [42] pelo autor

### 1.2.4. LLMs textuais completam textos

Como nos ensina o cientista da computação e polímata Stephen Wolfram [43], apesar de ser capaz de gerar textos que parecem ter sido escritos por humanos, na verdade o que o ChatGPT faz é essencialmente prever "a próxima palavra mais provável" (mais precisamente, o próximo *token* mais provável) para **completar uma frase**. Ou seja, **completar textos** com base em **instruções** (*Prompts*) fornecidas pelos usuários em linguagem natural.

The first thing to explain is that what ChatGPT is always fundamentally trying to do is to produce a “reasonable continuation” of whatever text it’s got so far, where by “reasonable” we mean “what one might expect someone to write after seeing what people have written on billions of webpages, etc.” [43].

A primeira coisa a explicar é que o que o ChatGPT está fundamentalmente sempre tentando fazer é produzir uma "continuação razoável" de qualquer texto que tenha recebido até então, onde por "razoável" queremos dizer "algo que alguém poderia esperar que um humano escrevesse depois de ter acesso ao que as pessoas escreveram em bilhões de páginas na web, etc." (tradução livre do autor de "*What Is ChatGPT Doing ... and Why Does It Work?*" de Stephen Wolfram) [43].

Os LLMs textuais são modelos *de Machine Learning* utilizados na IA Generativa, previamente treinados em vários idiomas inicialmente sem supervisão com quantidades massivas de dados de linguagem natural (artigos acadêmicos, livros, sites como o Wikipedia, conteúdos diversos da Internet). Após este pré-treinamento, os modelos são treinados novamente com outras fontes de dados mais confiáveis (menos dados, maior qualidade) e com supervisão humana (ajuste supervisionado, aprendizado por reforço).

Como veremos nos Capítulos 3 e 4, o treinamento de um LLM como o GPT-3 é realizado por redes neurais com arquitetura *Transformer*, capazes de realizar operações em paralelo (otimizando esforços computacionais), e assim o modelo consegue aprender padrões e estruturas existentes na linguagem natural. Concluído o treinamento, o modelo se torna bastante eficiente em.... **prever a próxima palavra em um texto.**

No exemplo adaptado de [43], imagine que o ChatGPT tem que completar a frase...

"A inteligência artificial é capaz de..."

Tendo sido treinado com grande parte do conteúdo da Internet, incluindo livros digitalizados, levando em conta todas as instâncias de frases parecidas com essa em apenas alguns segundos, o modelo de linguagem utilizado pelo ChatGPT é capaz de gerar uma lista de palavras (na verdade, *tokens*) com maior probabilidade de completar a frase...

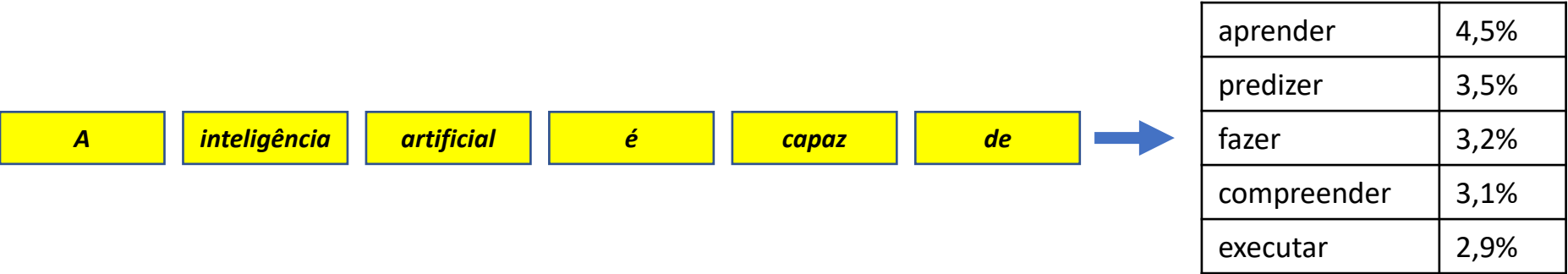




Figura 25 - Fonte: Adaptado de [43] pelo autor

E quando escreve (gera) um ensaio de 500 palavras, o ChatGPT faz isso repetidamente - *levando em conta o texto até agora, qual deve ser a próxima palavra mais provável?*

Embora o modelo procure pela próxima palavra (na verdade, *token*) mais provável para completar uma sentença, *nem sempre a palavra com maior probabilidade é a escolhida*, devido a uma certa aleatoriedade que pode ser controlada por hiperparâmetros como "temperatura" e pela amostragem **softmax**, como veremos na Seção 2.8. Assim, o texto é gerado com mais "criatividade", e quando passamos o mesmo contexto repetidamente podemos obter respostas diferentes em cada vez - *o que torna o texto ainda mais "humano"*.

 **You**

Por favor, complete a sentença "a inteligência artificial é capaz de" 5 vezes utilizando apenas um token em cada resposta, sem repetir nenhum token. Utilize apenas verbos no infinitivo. Mostre os resultados em formato de tabela, e indique a probabilidade (%) que foi gerada para cada token.

 **ChatGPT**

Nº	Frase	Probabilidade (%)
1	a inteligência artificial é capaz de aprender	23.4
2	a inteligência artificial é capaz de otimizar	18.9
3	a inteligência artificial é capaz de analisar	15.2
4	a inteligência artificial é capaz de prever	12.1
5	a inteligência artificial é capaz de facilitar	9.8





   

Figura 26

A princípio, a menos que seja instruído de forma diferente, o ChatGPT vai gerar vários *tokens* (um de cada vez) para completar uma sentença (parte superior da Figura 27).

Porém, a título de exemplo, podemos instruir o ChatGPT para simular o processo gerando frases onde adiciona apenas um *token* por vez (parte inferior da Figura 27).

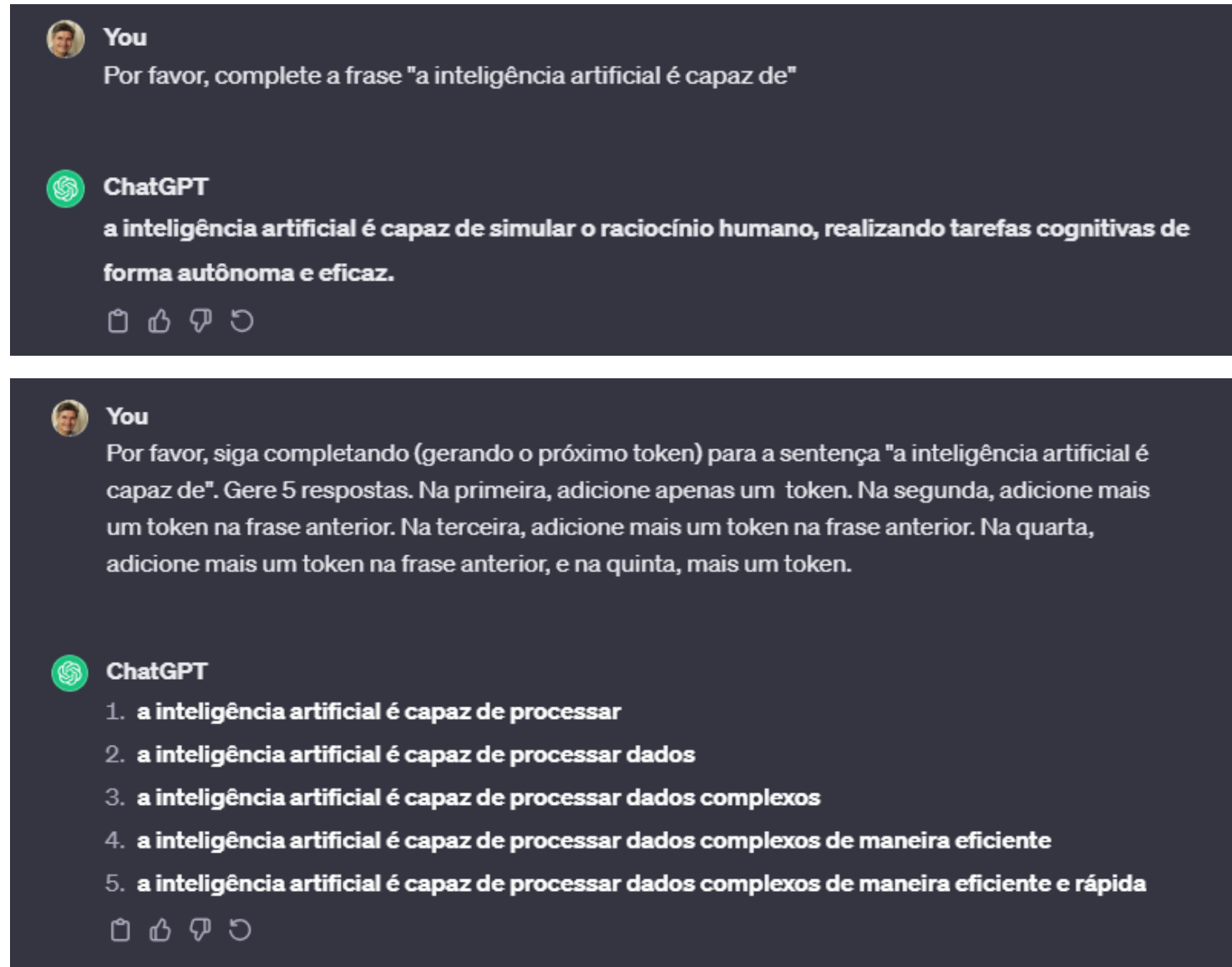


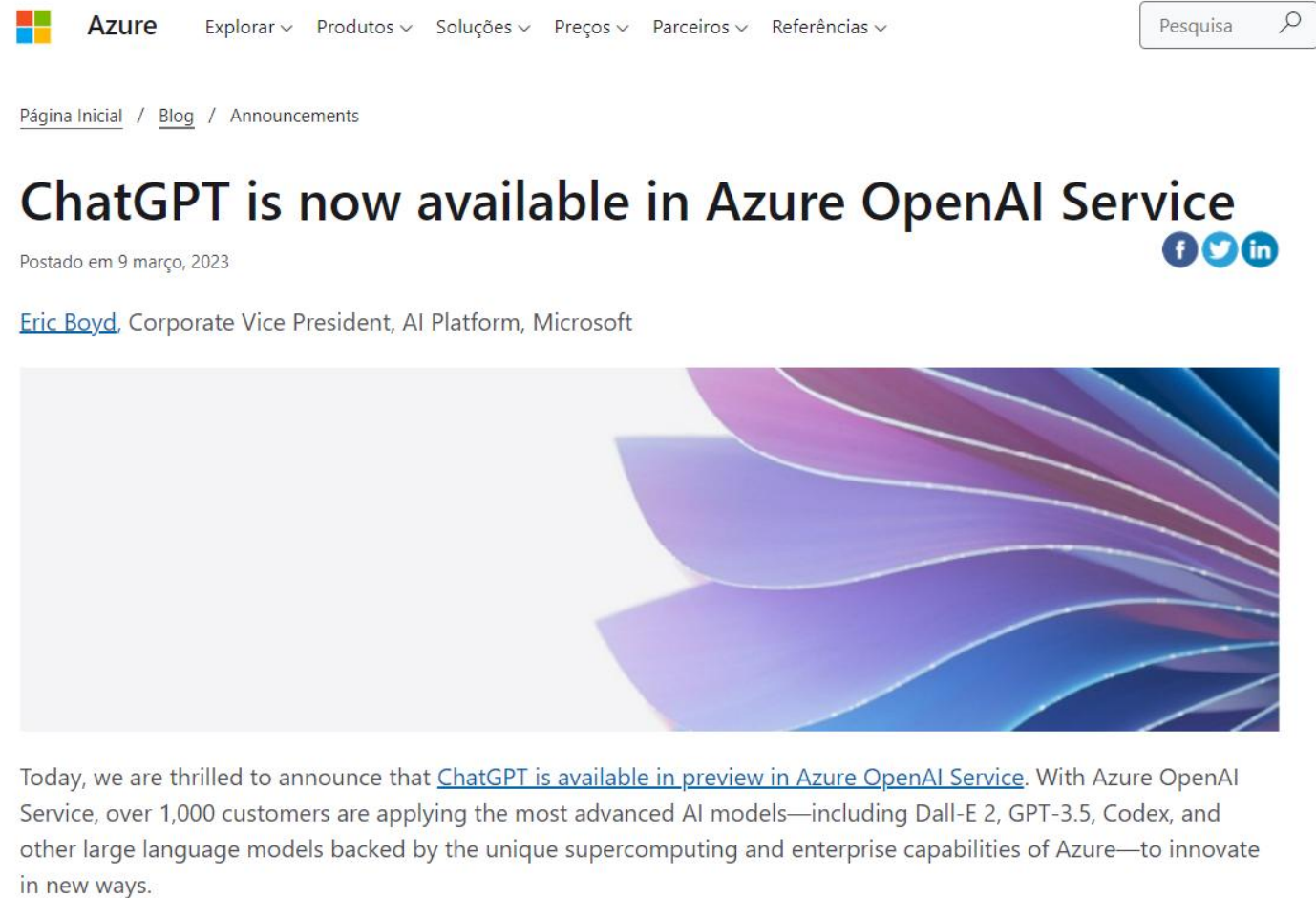
Figura 27



Faremos em seguida uma apresentação resumida e introdutória dos modelos de linguagem autorregressivos (que fazem a previsão do próximo *token* mais provável para completar uma sentença). Além destes, há também os chamados modelos de linguagem mascarados, que em vez de tentar completar uma sequência, tentam descobrir alguma palavra que está *faltando* no texto. Os interessados em uma abordagem mais completa e técnica podem consultar o excelente livro da Professora Aline Paes e colegas em [44].

### 1.2.5. MaaS (Model-as-a-Service)

Tal como anos atrás surgiu o modelo SaaS (*Software-as-a-Service*), ou o aluguel de software como um serviço, grandes empresas de IA e provedores de serviços de nuvem como a parceria da OpenAI com a Microsoft (através do Azure OpenAI), a *Amazon Web Services* (AWS) e a *Google Cloud Platform* (GCP) já vendem modelos de linguagem como serviços na modalidade MaaS (*Model-as-a-Service*), onde os clientes pagam para acessar via API os modelos mais avançados de IA Generativa como o DALL-E 2, o GPT-3.5 e o Codex da OpenAI.



The image is a screenshot of a blog post from the Azure website. At the top, the Azure logo is on the left, and a navigation menu with links like 'Explorar', 'Produtos', 'Soluções', 'Preços', 'Parceiros', and 'Referências' is on the right. A search bar labeled 'Pesquisa' is also present. Below the navigation, a breadcrumb trail reads 'Página Inicial / Blog / Announcements'. The main heading of the post is 'ChatGPT is now available in Azure OpenAI Service', with social media icons for Facebook, Twitter, and LinkedIn to its right. The post is dated 'Postado em 9 março, 2023' and is attributed to 'Eric Boyd, Corporate Vice President, AI Platform, Microsoft'. The post content begins with 'Today, we are thrilled to announce that [ChatGPT is available in preview in Azure OpenAI Service](#). With Azure OpenAI Service, over 1,000 customers are applying the most advanced AI models—including Dall-E 2, GPT-3.5, Codex, and other large language models backed by the unique supercomputing and enterprise capabilities of Azure—to innovate in new ways.' The background of the post features a decorative graphic of overlapping, wavy, translucent shapes in shades of purple and blue.

Figura 28 - Fonte: [45]

Por exemplo, diferentes versões do modelo GPT-3 da OpenAI (como a gpt-3.5-turbo ou a gpt-4-0314 utilizadas pelo ChatGPT) podem ser acessadas via chamadas de API.

A chamada da API pode ser limitada (por exemplo, quantidade de *tokens* no *Prompt*), e em geral se faz através de um *script* em Python ou outra linguagem. O *script* se conecta na API (após a devida autenticação, fornecendo chaves de acesso) e envia uma requisição em JSON, contendo as instruções (*Prompt*) para o que se deseja que o modelo faça.

A API envia a instrução para o modelo, que processa o *Prompt* e envia a resposta para a API, que então retorna a resposta para a aplicação cliente que fez a requisição (Figura 29).

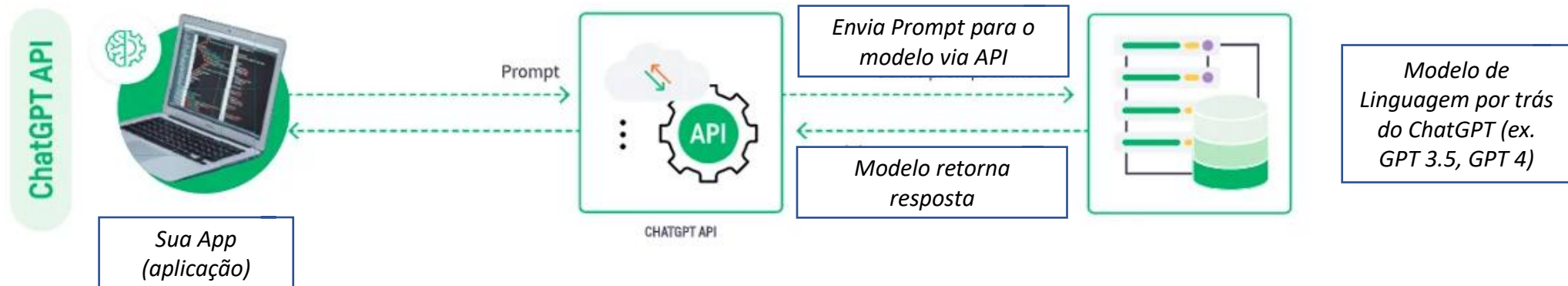


Figura 29 - Fonte: Adaptado de [46]

Muitas organizações estão integrando o ChatGPT (via API) com suas próprias plataformas e aplicações, que depois de "turbinadas" pelo ChatGPT (ou outra IA Generativa) são revendidas para outros clientes.

As grandes empresas de consultoria já se organizaram para oferecer um novo serviço para as organizações: ajudar seus clientes a utilizarem o ChatGPT internamente, para melhoria de processos e redução de custos.

# THE WALL STREET JOURNAL.

[Subscribe](#)[Sign In](#)

English Edition ▾ | [Print Edition](#) | [Video](#) | [Audio](#) | [Latest Headlines](#) | [More ▾](#)

[World](#) [Business](#) [U.S.](#) [Politics](#) [Economy](#) [Tech](#) [Finance](#) [Opinion](#) [Arts & Culture](#) [Lifestyle](#) [Real Estate](#) [Personal Finance](#) [Health](#) [Science](#) [Style](#) [Sports](#)



CIO JOURNAL

## Consultants Emerge as Early Winners in Generative AI Boom

Lacking in-house know-how, companies are turning to outside experts for help putting ChatGPT-like tools to work

By [Angus Loten](#) [Follow](#)

June 22, 2023 5:12 pm ET

Figura 30 - Fonte: [47]

Saiba mais sobre a corrida desenfreada (e até certo ponto divertida, pelo grau de oportunismo) das grandes consultorias para "ajudar seus clientes" a utilizar o ChatGPT e outras IAs Generativas internamente, para melhorar seus processos de negócio em [48] (*accenture*), [49] (EY) e [50] (Deloitte).

O fato é que *um enorme ecossistema de negócios está se formando em torno da IA Generativa*, e o prognóstico é que as organizações *de qualquer ramo de atividade ou porte* que não se adaptarem ao novo paradigma que a IA está trazendo em poucos anos estarão em séria desvantagem competitiva, terão menor eficiência operacional, verão seu *market share* se reduzir com a perda de clientes e oportunidades e terão dificuldades em reter mão de obra e contratar novos talentos. Será? Vamos perguntar ao ChatGPT.

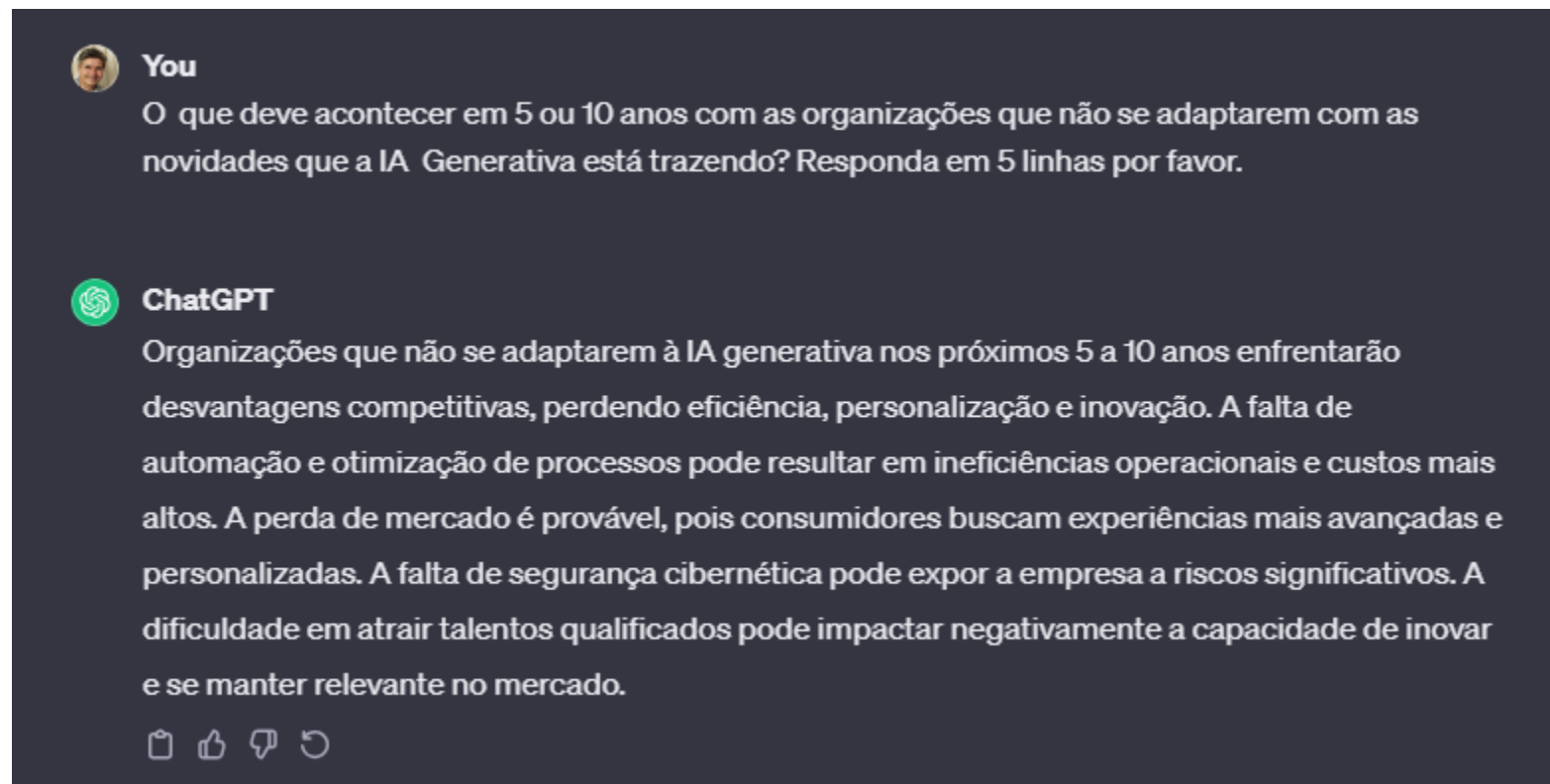


Figura 31

### 1.2.6. Customizando modelos de linguagem

É praticamente inviável para as organizações criarem do zero os seus próprios LLMs (do porte de um GPT-3), pois o custo do esforço computacional para treinar estes modelos pode alcançar milhões de dólares. Como veremos, a OpenAI utilizou dezenas de milhares de GPUs para treinar o GPT-3, e mesmo assim o processo demorou meses. O treinamento do GPT-4 foi certamente ainda mais intenso no consumo de recursos. Assim, apenas algumas poucas *Big Techs* têm os recursos (e o conhecimento técnico) necessários para criar e treinar modelos deste porte.

Já que de modo geral não é viável para as organizações treinar os seus próprios LLMs, podem utilizar os modelos pré-treinados como serviços na nuvem acessíveis via API, como vimos. Além de utilizar os modelos originais, há também a opção de tentar customizar os grandes modelos de propósito geral, fazendo alguns ajustes finos para melhorar sua precisão em domínios específicos.

Por exemplo, a parceria da Microsoft com a OpenAI [51, 52] permite que alguns modelos da OpenAI previamente treinados sejam "customizados" (ou melhor, ajustados) pelo cliente para que tenham maior precisão em domínios específicos (por exemplo, medicina, ou algum outro ramo onde a precisão é mais importante que a generalidade).

*Porém, há limitações no ajuste de grandes modelos.*

Embora seja possível fazer ajustes em alguns modelos da OpenAI como o gpt-35-turbo-0613, o babbage-002 e o davinci-002 para que sejam mais focados e precisos em um ramo de conhecimento específico [53], este recurso *ainda é limitado para apenas algumas regiões do mundo*, e o custo do treinamento (recursos computacionais) ainda é elevado.

Uma outra opção já ao alcance de várias organizações é buscar a customização de modelos menores [54].

Em vez de utilizar um LLM gigante treinado com dados genéricos da Internet usando centenas de bilhões de parâmetros a um custo computacional imenso, em muitos casos é possível utilizar um modelo menor, com menos parâmetros, e com dados mais confiáveis para um certo domínio (menor quantidade, maior qualidade).

"Todos os grandes modelos que você obtém de provedores externos são treinados com dados retirados da Internet. Porém, dentro da sua organização você tem muitos conceitos e dados internos que estes modelos desconhecem."

"All the large models that you can get from third-party providers are trained on data from the web. But within your organization, you have a lot of internal concepts and data that these models won't know about."

Matei Zaharia, Cofounder and Chief Technology Officer, Databricks, and Associate Professor of Computer Science, University of California, Berkeley

Figura 32 - Fonte: [54]

Além dos dados, os próprios parâmetros utilizados no treinamento dos modelos pode ser ajustados (o que altera os pesos utilizados nas camadas das redes neurais). De modo geral é bem mais barato ajustar apenas algumas pequenas partes do modelo, o que pode ser feito com o uso de bibliotecas como a PETF (*Parameter Efficient Fine Tuning*) [55] e métodos de treinamento de LLMs que consomem menos recursos computacionais como o LoRA [56]. A distinção entre os *modelos proprietários* (como os modelos GPT da OpenAI) e os *modelos open source* também é importante. Por exemplo, o já mencionado LLaMA é um modelo de linguagem aberto (não comercial) e de alta performance. Este porém é um domínio que embora esteja se tornando mais acessível ainda é bastante técnico e requer conhecimento especializado. Para os modelos STF ainda é factível tentar este tipo de "ajuste fino", mas para os modelos RLHF (como o GPT-3) este assunto ainda está em estágio de pesquisa (os termos "STF" e "RLHF" serão explicados no Capítulo 3).

*"Modelos menores, entretanto, oferecem uma alternativa viável. "Eu acredito que vamos nos afastar de 'Eu preciso de meio trilhão de parâmetros no modelo' para 'talvez eu precise de 7, 10, 30 ou 50 bilhões de parâmetros nos dados que eu realmente possuo'... A redução na complexidade vem quando se reduz o foco de modelos de propósito geral que envolvem todo o conhecimento humano para um domínio específico do conhecimento, com dados de alta qualidade relevantes para o seu caso, pois é isso que as organizações realmente precisam".*

Smaller models, however, provide a viable alternative. "I believe we're going to move away from 'I need half a trillion parameters in a model' to 'maybe I need 7, 10, 30, 50 billion parameters on the data that I actually have,'" says Carbin. "The reduction in complexity comes by narrowing your focus from an all-purpose model that knows all of human knowledge to very high-quality knowledge just for you, because this is what individuals in businesses actually really need."

Figura 33 - Fonte: [54]

Ainda no contexto de "ajuste de modelos" para maior adaptação a um domínio, o serviço CustomGPT [57] permite integrar um bot com um website ou plataforma, fazer a ingestão do conteúdo do seu website e indexar. O bot vai então dar suporte sobre este conteúdo, fornecendo aos usuários respostas mais acuradas e relevantes.

Os arquivos que o CustomGPT importa para indexar incluem PDF, Microsoft Office, Google Docs, arquivos de texto, áudios e vídeos, e todo o conteúdo é criptografado e acessado de forma segura. Depois que o seu bot é criado, é possível fazer algumas customizações, por exemplo, configurar o bot para responder perguntas levando em conta apenas o conteúdo que você indicou, ou levando em conta tanto o seu conteúdo próprio quanto a base do ChatGPT (no caso, o modelo GPT-4).



# Capítulo 2 - Tokenização e *Word Embeddings*

Neste capítulo, veremos algumas abordagens que evoluíram ao longo dos últimos anos na tentativa de resolver os difíceis desafios da modelagem da linguagem. Veremos que as primeiras tentativas importantes adotavam a hipótese da *semântica distribucional* e foram baseadas em *modelos de linguagem probabilísticos* e a sua utilização de n-gramas, ou sequências de *tokens* formados por *n* caracteres adjacentes. Veremos que apesar de serem fáceis de entender, estes modelos não "escalam", ou como melhor explicado em [44], sofrem da “maldição da dimensionalidade”. Aqui, já surge o conceito de *token*, que é bastante importante.

Como uma alternativa mais eficiente, surgiram os modelos de linguagem baseados em redes neurais. Aqui, a ideia é treinar uma rede neural para que ela se aproxime de uma função matemática capaz de prever a probabilidade de ocorrência de uma palavra para completar uma sequência com outras palavras, passada como contexto (Figura 34). Além de completar frases, outras tarefas (como tradução de sentenças de um idioma para outro) também podem ser executadas por estas redes. Uma vez treinada, a rede neural é capaz de *estimar qual é a próxima palavra mais provável*, considerando um contexto de palavras anteriores.

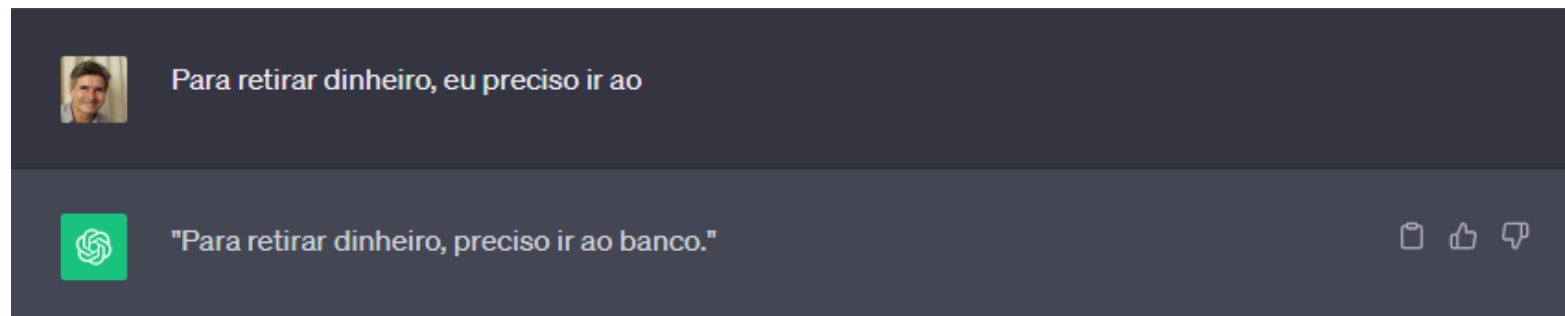


Figura 34

A grande vantagem dos modelos neurais é que de posse de uma função para estimar probabilidades, o modelo poderá ser aplicado para prever a probabilidade de *palavras que não estavam disponíveis durante o treinamento*. A discussão dos modelos neurais nos levará ao conceito de *Word Embeddings*.

De 2017 a 2020, a necessidade de criar *Embeddings* contextualizados e superar outras limitações técnicas impulsionou o desenvolvimento das Redes Neurais Recorrentes (RNN - *Recurrent Neural Networks*). As RNNs podem trabalhar com qualquer tipo de dado sequencial, o que inclui textos e também áudio, vídeo ou mesmo códigos de computador. Falaremos mais sobre as RNNs na Seção 4.2.

Na esteira das RNNs foram melhorados os métodos de *Word Embedding*, com o surgimento por exemplo do método ELMo (*Embeddings from Language Models*), além dos *Transformers*, seguidos pelo BERT (*Bidirectional Encoder Representations from Transformers*) e pelos primeiros modelos GPT (*Generative Pre-trained Transformer*) lançados pela OpenAI por volta de 2020.

Aqui, após a discussão sobre tokenização e *Word Embeddings*, vamos encerrar o Capítulo 2, mas a história continua:

- Uma evolução importante foi a criação dos **mecanismos de Atenção** nas RNNs. Esta técnica permitiu modelar as dependências (semânticas e contextuais) mesmo entre palavras (*tokens*) que estavam distantes na sequência, em particular, pelas RNNs mais avançadas com mecanismos LSTM (*Long Short Term Memory*).
- Apesar da evolução importante representada pelas Redes Neurais Recorrentes, estas redes ainda operavam de forma inerentemente sequencial, e isso criava problemas de escalabilidade computacional, no treinamento de grandes modelos de linguagem. Além disso, como veremos as RNNs sofrem do chamado "problema do gradiente", uma dificuldade técnica que se manifesta em redes neurais com muitas camadas (*Deep Learning*).

- Como superação para os problemas das RNNs, surgiu finalmente a grande jóia da coroa, que são as redes neurais com arquitetura *Transformer* baseadas puramente em mecanismos de Atenção, sem as recorrências ou convoluções características das RNNs e suas variações, e que tornaram possível que os bilhões de cálculos realizados pela rede durante o treinamento dos modelos fossem realizados *em paralelo*.
- Isso por sua vez viabilizou o treinamento dos grandes modelos (que ainda assim demora vários meses, utilizando milhares de GPUs especializadas, e ainda custa milhões de dólares), e em um sentido prático, tornou possível, ou acelerou de forma importante o surgimento do ChatGPT.

O treinamento dos grandes modelos de linguagem será nosso tema no Capítulo 3, e as redes com arquitetura *Transformer* e seus mecanismos de Atenção serão o assunto principal do Capítulo 4.

## 2.1. Semântica Distribucional

Se desejamos criar modelos de linguagem, é preciso levar em conta que na linguagem natural além da sintaxe [58] existe a semântica [59] - o estudo do *significado* das palavras. Lidar com a sintaxe e as regras lexicais (como combinar caracteres para formar palavras válidas em um idioma qualquer) é uma parte do problema, mas um desafio ainda mais difícil é como representar a semântica em um modelo computacional.

Há diferentes definições de "semântica" e "significado", mas para nossos propósitos podemos nos alinhar com o pensamento de Wittgenstein em sua filosofia da linguagem [60], no qual *o significado de uma palavra é derivado da forma como ela é utilizada*, ou seja, a produção de significados ocorre dentro de um contexto. Por exemplo, os diferentes significados do termo "banco" em "Para retirar dinheiro eu preciso ir ao banco" e "O rapaz se encontrou com a moça na praça e eles se sentaram juntos no banco" variam considerando os demais termos próximos nas sentenças, ou seja, *conforme o uso*.

É esta a mesma abordagem que é utilizada nos modelos de linguagem usados em *Machine Learning*. No caso, o treinamento é baseado na *Semântica Distribucional* [44, 61], que abrange pesquisas e métodos para categorizar e quantificar semelhanças semânticas (isto é, de significado) entre termos da linguagem natural, com base em *propriedades distribucionais* destes termos, extraídas de grandes *Datasets*.

De forma simples, a Semântica Distribucional é a hipótese de que o significado de uma palavra em uma frase pode ser inferido através do contexto fornecido pelas palavras próximas. Este pressuposto permite que as redes neurais modernas calculem a probabilidade de uma determinada palavra (tecnicamente, *token*) ser "a mais provável" para completar uma frase.

A ideia básica é baseada na seguinte hipótese:

"Termos da linguagem natural com distribuições semelhantes têm significados semelhantes".

ou, dito de outra forma,

"Palavras diferentes que ocorrem geralmente em um mesmo contexto tendem a ter significados semelhantes, ou algum outro tipo de similaridade contextual".

A frequência com que palavras diferentes são utilizadas próximas em documentos também indica que há alguma proximidade entre elas, ainda que não tenham o mesmo significado. Por exemplo, em muitos documentos as palavras "tubarão", "animal" e "perigoso" são utilizadas em um mesmo contexto, dado que no mundo real sabemos que os tubarões são animais perigosos, e isso obviamente se reflete na linguagem natural.

Da mesma forma as palavras "homem" e "soldado" tentem a ocorrer mais próximas (em contextos típicos da linguagem natural) do que, por exemplo, as palavras "mulher" e "caminhoneiro", embora nem todo homem seja soldado e mulheres possam ser soldados e dirigir caminhões.

O que o método sugere é que, considerando as distribuições estatísticas do uso das palavras nas frases da linguagem natural, deve haver (estatisticamente) *maior proximidade* entre "tubarão" e "animal" do que entre "tubarão" e "caminhoneiro" por exemplo. Tubarões não dirigem caminhões fora de contextos ficcionais, de modo que os domínios de uso destes dois termos são muito afastados (bem, artigos podem relatar que um caminhoneiro foi morto por um tubarão, mas ainda assim *estatisticamente* a proximidade entre estes termos é baixa).

Como veremos a seguir, os modelos de linguagem probabilísticos baseados em n-gramas (BOX 2) eram puramente estatísticos.

## 2.2. Modelos de Linguagem Probabilísticos

Simplificando bastante (convido os interessados em uma abordagem mais técnica a consultar [44]), um modelo de linguagem probabilístico "atribui uma probabilidade a uma sequência de palavras". Uma tarefa típica para estes modelos é "predizer a próxima palavra mais provável em uma sequência, condicionando-a às palavras precedentes".

No exemplo da Figura 35 extraído de [44], um modelo de linguagem probabilístico foi utilizado para prever as probabilidades de palavras, considerando um contexto anterior.

Por exemplo, considerando "A" e "casa" como palavras prévias, o modelo calcula que há uma probabilidade de 58% da próxima palavra na sequência ser "caiu" (formando a sentença "A casa caiu"). Já se as duas palavras precedentes são "A" e "parede", o modelo prediz 65% de probabilidade para a palavra "verde".

Estas probabilidades são calculadas com base na frequência de ocorrência das palavras na linguagem natural, considerando um determinado *Dataset* com milhares de palavras chamado **vocabulário**. Mais detalhes sobre como estas probabilidades são geradas são fornecidos no BOX 2 e na Seção 2.4.

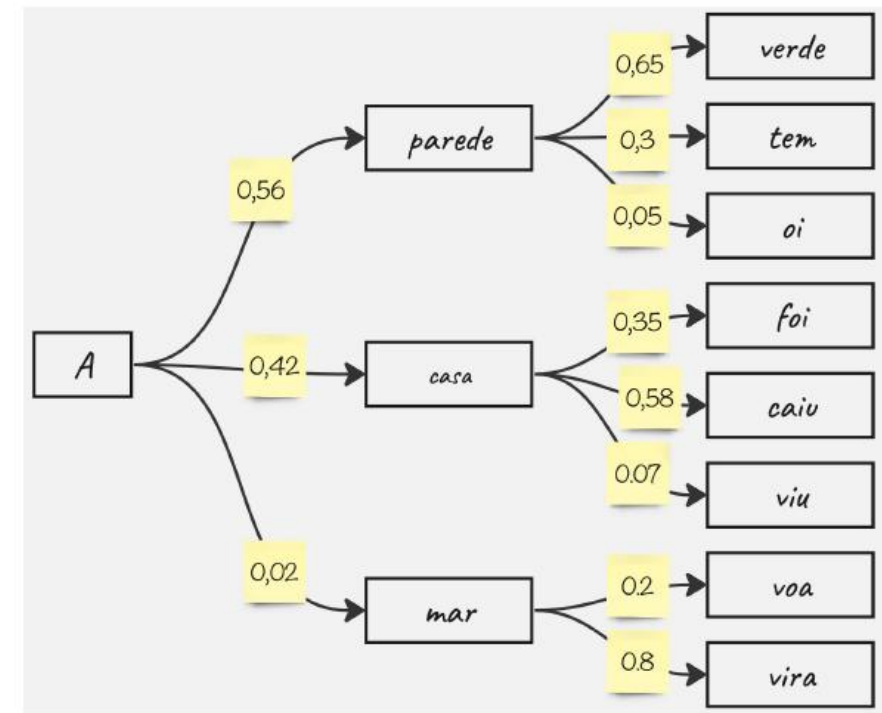


Figura 35 - Fonte: [44]

Nos exemplos anteriores, é importante ressaltar que estamos falando de *probabilidades condicionais*, ou do cálculo da probabilidade de uma palavra qualquer (em um vocabulário bem definido) ser a *próxima mais provável* em uma sentença, *considerando algumas palavras que vieram antes*, ou uma *janela de contexto*.

Estes tipos de modelos de linguagem cuja principal tarefa é "completar um texto com a próxima palavra mais provável" são chamados *autorregressivos*, e vamos apresentar uma definição formal para eles na Seção 2.9.

Na verdade, em vez de palavras (no sentido usual do termo), os modelos de linguagem probabilísticos utilizam sequências de *tokens* chamadas n-gramas (BOX 2). Estes *tokens* podem ser caracteres, partes de palavras, palavras inteiras ou mesmo grupos de palavras, com veremos oportunamente, mas por hora vamos nos referir a "palavras" em favor da simplicidade.

Embora simples de entender, os modelos probabilísticos têm limitações. Por exemplo, o modelo precisa calcular uma probabilidade para cada palavra que existir no vocabulário, e estamos falando de probabilidade condicional, onde é preciso levar em conta uma sequência anterior de palavras para que a probabilidade da *próxima* seja calculada. Isso cria dificuldades computacionais rapidamente. Como mencionado em [44], para modelar uma sequência prévia de 10 palavras e um vocabulário com 100.000 palavras, o modelo teria que ter  $100.000^{10} - 1$  parâmetros (este problema é chamado de a "maldição da dimensionalidade").

Uma tentativa de se livrar da "maldição da dimensionalidade" é adotar a *suposição de Markov* de que "apenas o passado mais recente é importante para o futuro", ou seja, levar em conta sequências prévias menores no cálculo das probabilidades. Por exemplo, em um modelo de bigramas (BOX 2) podemos "ignorar o passado mais distante" e levar em conta apenas *uma* palavra precedente, além da própria palavra cuja probabilidade se deseja calcular. Em um modelo de trigramas, levaríamos em conta *duas* palavras precedentes, e assim por diante.

Porém, não há lanche grátis, e ao se fazer a escolha por considerar contextos anteriores menores, ignorando algumas palavras do contexto precedente para facilitar os cálculos de probabilidade condicional da *próxima* palavra da sequência, perde-se informação, e assim a precisão do modelo diminui.

Intuitivamente, podemos compreender que o modelo provavelmente terá mais precisão para eleger "Suíça" como a palavra mais provável para completar a sentença...

"É muito bom esquiar nos Alpes da ..."

... desde que *todas* as palavras precedentes (o contexto "É", "muito", "bom", "esquiar", "nos", "Alpes", "da") sejam levadas em conta.

Se o um modelo de linguagem probabilístico utilizar um método de bigrama, tentando calcular a probabilidade da próxima palavra em questão ("Suíça") levando em conta uma *única* palavra precedente, teríamos...

"da"... ???

Há outros problemas associados aos modelos probabilísticos de linguagem.

Se o modelo escolher *sempre* a palavra de maior probabilidade (considerando todo o vocabulário) para ser a próxima de uma sequência, haverá falta de criatividade ou diversidade nas respostas, e a saída do modelo ficará menos natural (para seres humanos).

Como veremos, este problema é resolvido nas abordagens mais modernas com redes neurais através de hiperparâmetros de "temperatura".



## BOX 2. N-gramas

Um n-grama [62] é uma sequência de *tokens* formados por **n** caracteres adjacentes (incluindo espaços e sinais de pontuação), sílabas, ou por palavras completas em um banco de dados de palavras de um idioma (*language dataset*). Um n-grama de tamanho 1 é chamado unigrama, de tamanho 2 é um bigrama etc.

No caso em que a unidade do n-grama é um único caractere, uma sentença como "Ser ou não ser" pode ser representada por 13 diferentes combinações de dois caracteres ou bigramas (um traço representa um espaço): "Ser ou não ser" -> bigramas de caracteres = "Se", "er", "r\_", "\_o", "ou", "u\_", "\_n", "nã", "ão", "o\_", "\_s", "se", "er". Já se considerarmos unigramas de palavras completas, "Ser ou não ser" vai gerar 4 unigramas: "Ser", "ou", "não", "ser".

Há ainda um outro significado para n-grama no contexto dos modelos de linguagem probabilísticos [62, 63]:

Um modelo de linguagem n-grama é um modelo puramente estatístico onde se assume a premissa de que "a probabilidade de ocorrência de uma próxima palavra em uma sequência" depende mais fortemente de uma quantidade fixa de palavras prévias" (o futuro é mais influenciado pelo passado mais recente).

Se apenas uma palavra prévia é considerada na previsão da próxima, é chamado um modelo bigrama, se forem consideradas duas palavras, é um modelo trigrama, e de modo geral, se forem levadas em conta  $n-1$  palavras anteriores, é um modelo n-grama.

Por exemplo, em "Eu comprei um carro vermelho" a probabilidade de ocorrência da palavra "carro" levaria em conta em um modelo bigrama as duas palavras imediatamente precedentes "comprei um", e a probabilidade de "vermelho" levaria em conta as precedentes "um carro".

O cálculo das probabilidades leva em conta a frequência de ocorrência real das palavras em cada idioma (quais palavras são mais comuns na linguagem), considerando *datasets* imensos como vocabulário. Mas isso gera um problema - na medida em que  $n$  aumenta, a quantidade de diferentes combinações entre n-gramas se torna imensa, e portanto é impraticável deduzir as probabilidades de todas as combinações possíveis.

Além disso, embora possuam aplicações importantes, uma dificuldade com os n-gramas é a sua incapacidade de representação explícita de dependências de longo alcance entre palavras.

Assim, embora os n-gramas sejam adequados para certos tipos de tarefas em processamento de linguagem, sofrem de limitações importantes em casos onde há necessidade de generalização, ou a capacidade do modelo em encontrar a próxima palavra mais provável para completar uma sentença mesmo nos casos onde os exemplos passados como entrada para o modelo contenham palavras que *não estavam presentes* nos dados utilizados durante o treinamento (estão fora de um vocabulário fixo).

Para resolver este problema da generalização, é preciso treinar o modelo de linguagem de forma que ele se aproxime de alguma *função matemática* que seja capaz de calcular as probabilidades nestas circunstâncias - calculando probabilidades mesmo quando o contexto recebido como entrada não fazia parte dos dados do treinamento - e isso nos leva aos modelos de linguagem neurais, ou baseados em redes neurais.

## 2.3. Modelos de Linguagem Neurais

Suponha que a safra de tomates em toneladas ( $y$ ) tem uma correlação linear com a quantidade de chuvas em uma região ( $x$ ). Neste caso, pode-se tentar encontrar uma função que permita estimar qual será a quantidade da safra, dada uma certa quantidade de chuvas na região em um certo período.

Esta função matemática seria um exemplo de *modelo*. Uma das formas de se encontrar esta função é utilizando *Machine Learning*, treinando modelos com algoritmos de regressão linear, de modo supervisionado, com uma certa quantidade de dados. Uma vez encontrada uma função que tenha boa performance em suas predições, espera-se que o modelo seja *capaz de fazer generalizações* e prever a safra de tomates mesmo para dados de quantidades de chuva que *não* estavam presentes no *dataset* utilizado no treinamento.

Esta ideia também é aplicável ao problema de "encontrar uma função matemática capaz de calcular a probabilidade de uma palavra ser a mais provável para completar uma sequência prévia de palavras (contexto), considerando um vocabulário com dezenas de milhares de palavras". Neste caso, a função que procuramos é bem mais complexa, mas ainda assim pode ser (teoricamente) aproximada com o uso de *Machine Learning*, mais precisamente, por *Deep Learning*, ou seja, com modelos treinados por redes neurais profundas, com várias camadas ocultas.

Como veremos (Seção 4.1), durante o treinamento de um modelo deste tipo, os pesos nas diversas camadas da rede neural são continuamente reajustados em diversas iterações, visando sempre reduzir os erros cometidos pelo modelo. Com o tempo, o modelo vai aprendendo a estrutura da linguagem, e vai cometendo menos erros, se tornando cada vez mais capaz (= melhor performance) de prever a próxima palavra para completar uma sentença (ou realizar outros tipos de tarefa de linguagem natural, como analisar sentimentos em textos ou fazer traduções por exemplo).

Como veremos, há muitos requisitos técnicos para este tipo de treinamento de modelos de linguagem. Um requisito importante é que tudo que será passado como entrada (INPUT) para a rede precisa ser representado de forma numérica, já que redes neurais só conseguem trabalhar com números. Assim, o primeiro passo do processo é a tokenização - as palavras de um vocabulário são convertidas em *tokens*, utilizando um algoritmo especial. Estes tokens são identificados por números inteiros. Vamos discutir a tokenização na Seção 2.5.

Antes de serem processados, os *tokens* precisam ser *vetorizados* (representados por vetores com números reais). Para que o modelo de linguagem complete as sentenças com palavras de forma que faça sentido para nós humanos, é preciso mais do que entender a sintaxe ou a estrutura da linguagem. O *relacionamento entre as diferentes palavras* em uma sentença, seja semântico ou contextual, também precisa ser levado em conta, dado que mesmo palavras que aparecem no início de uma frase podem ser importantes para compreender o significado de uma outra que aparece mais distante. Para que esta importante informação - o contexto, ou a similaridade entre diferentes palavras - também possa ser levada em conta no treinamento do modelo, foram desenvolvidos métodos de "*Word Embeddings*" contextuais que representa os *tokens* por vetores numéricos, sobre os quais vamos conversar na Seção 2.6.

Deixando de lado por hora os aspectos mais técnicos, a ideia que desejamos passar aqui é que os modelos de linguagem neurais são, como o nome sugere, treinados por redes neurais. Suas sequências de entrada (contextos) são tokenizadas, e em seguida, vetorizadas (transformadas em vetores com números reais), com um método de "*Embedding* Posicional". A rede neural é treinada com uma quantidade enorme de textos (para que possa aprender sobre as palavras mais frequentes, e a estrutura da linguagem), e tem como missão encontrar uma função matemática (aproximada) que permita prever "qual é a próxima palavra mais provável" para completar (ou traduzir) uma sentença, levando em conta um contexto com algumas palavras anteriores, *com generalização*, ou seja, mesmo que a sequência passada contenha palavras que *não* estavam disponíveis durante o treinamento. Estas redes no início eram Redes Neurais Recorrentes (RNN - *Recurrent Neural Networks*) [64], que evoluíram para RNNs com mecanismos de Atenção, e finalmente para o estado da arte atual, que são as redes neurais com arquitetura *Transformer* (Capítulo 4).

## 2.4. De onde vêm as probabilidades?

Mencionamos que o modelo de linguagem utilizado pelo ChatGPT escolhe a próxima palavra (*token*) para completar uma sentença "baseado em probabilidades". De onde vêm estas probabilidades? Da linguagem natural, tal como se apresenta nos bilhões de documentos que foram utilizados para treinar o modelo.

Considerando todas as palavras (*tokens*) - todo o vocabulário utilizado no treinamento do modelo em um certo idioma (por exemplo, Inglês) - quais são as combinações de caracteres mais comuns? Por exemplo, a combinação de caracteres (*token*) "at" aparece em "cat" (gato), "bat" (morcego) e também é um pronome muito utilizado em Inglês que indica lugar ou tempo (em, na). Já a combinação "do" aparece em inúmeras palavras como "dog" (cachorro) e também é o nome de um verbo (fazer).

Estes são portanto pares de letras (bigramas) mais frequentes, e portanto, mais prováveis de serem utilizados em textos na linguagem natural em Inglês do que, por exemplo, o par "yt" que também é encontrado em várias palavras (como "hypnotize") mas não é tão frequente como "at" ou "do".

Um algoritmo pode contar a ocorrência destes pares de caracteres em milhões de textos escritos em Inglês, e gerar uma distribuição estatística da probabilidade de ocorrência de qualquer par de caracteres (ou de apenas um caractere, ou de três etc.). Com uma quantidade suficiente de textos em Inglês é possível obter estimativas muito boas não apenas da probabilidade de pares de letras, mas para três ou quatro caracteres que apareçam em sequência, ou mesmo para palavras inteiras (*tokens* com n caracteres, ou n-gramas) [43]. Existem cerca de 40.000 palavras que são mais utilizadas em Inglês, e analisando um *dataset* com milhões de livros, contendo bilhões de palavras, é possível obter estimativas muito boas sobre o quão comum cada palavra é.

Com base apenas nisso, podemos pedir a um programa de computador para gerar frases aleatórias. Por exemplo, gerando palavras com quantidades de caracteres progressivamente maiores e que tenham maior probabilidade de ocorrência, é possível obter frases que se tornam cada vez mais "realistas", como no exemplo de [43] (em Inglês):

5 special average vocab consumer market prepara injury trade consa usually speci utility

Figura 36 - Fonte: [43]

Na frase aleatória acima, cada palavra foi gerada isoladamente em função de seu maior uso no idioma Inglês (maior probabilidade). Embora a frase ainda não faça sentido, nela já é possível reconhecer várias palavras comuns. Para obter resultados ainda melhores, em vez de considerar apenas palavras isoladas, podemos levar em conta as probabilidades de ocorrência de pares de palavras, ou pares de *tokens* com sequências de caracteres (n-gramas) mais longas. A frase seguinte foi gerada desta forma. Ainda não faz sentido mas já parece um pouquinho menos aleatória que a anterior.

cat on the theory an already from a representation before a

Figura 37 - Fonte: [43]

O problema com este método de modelagem "puramente estatística" é que mesmo considerando um **vocabulário** contendo apenas as 40.000 palavras mais comuns em Inglês existem 1.6 bilhões de combinações possíveis de sequências de dois caracteres (bigramas). Para três caracteres (trigramas) o número de combinações já aumenta para 60 trilhões!

Além disso, como mencionado, o vocabulário podem estar incompleto, com palavras raras ausentes.

Estes problemas da modelagem de linguagem puramente estatística de n-gramas - baseada unicamente na frequência de ocorrência de palavras e que não leva em conta as relações semânticas (significados) - foram superados com o uso de redes neurais para treinar modelos de linguagem que são capazes de estimar as probabilidades nas quais diferentes sequências de caracteres (*tokens*) podem ocorrer (em algum idioma), *mesmo que aquelas sequências jamais tenham sido utilizadas em qualquer texto já escrito* (ou seja, podem prever palavras que não estavam presentes nos dados de treinamento).

Estes são os “grandes modelos de linguagem” como o GPT-3 da OpenAI (e vários outros) - modelos pré-treinados que são capazes de calcular estas probabilidades, e este é o processo utilizado “por trás dos panos” pelo ChatGPT.

Como veremos mais adiante com algum detalhe, as palavras dos textos com instruções que passamos para o ChatGPT são convertidas para *tokens*, que em seguida são vetorizados (representadas por vetores numéricos de “*Embedding*” posicional). O modelo de linguagem já treinado que o ChatGPT utiliza (como o GPT-3.5 ou o GPT-4) processa estes vetores de “*Embeddings*” e gera as probabilidades para as palavras que possam completar o texto passado. Ou seja, a cada vez que precisa gerar uma nova palavra para completar uma frase, o modelo de linguagem do ChatGPT gera uma lista de números que indicam as probabilidades para cada uma das cerca de 50.000 palavras (*tokens*) que existem no vocabulário de *tokens* utilizado no treinamento do modelo, e que poderiam completar o texto. Calculadas todas estas probabilidades, os *tokens* mais prováveis são utilizados para completar a sentença (embora nem sempre seja escolhido exatamente o *token* com MAIOR probabilidade), e deste modo o ChatGPT vai gerando seus textos, *token por token*.

Como podemos imaginar, para gerar todas estas probabilidades e indicar “a próxima palavra mais provável” para completar um texto, *token por token*, a rede neural utilizada pelo modelo do ChatGPT (que tem bilhões de parâmetros ou pesos ajustáveis) precisa fazer uma quantidade colossal de cálculos, e ainda assim o Assistente fornece suas “respostas” quase em tempo real, o que é um testemunho da grande eficiência das soluções adotadas.

Para que tudo isso funcione, o treinamento do modelo de linguagem é crucial. Durante o treinamento do modelo, os pesos da rede neural são ajustados para reduzir erros até o ponto em que o modelo - a "função matemática" que calcula estas probabilidades - alcança uma performance satisfatória, e se torna bom o suficiente para "completar textos" - *sobretudo em novos textos que não estavam presentes no Dataset utilizado no treinamento*. Afinal, como diz Pedro Domingues [65], "em *Machine Learning* é a capacidade de generalizar para além dos exemplos disponíveis nos dados de treinamento que conta". *Voilà*. Depois de treinado, o modelo utilizado pelo ChatGPT é capaz até mesmo de *inventar novas palavras*, como nomes para Pokemons (Figura 38).

Para entender melhor *como* os modelos de linguagem são treinados e, em seguida, *como* estas probabilidades para os *tokens* mais prováveis são geradas em tempo real pelos modelos, com tão boa performance, precisamos nos familiarizar com os conceitos já mencionados de tokenização, vetorização e "*Word Embeddings*".

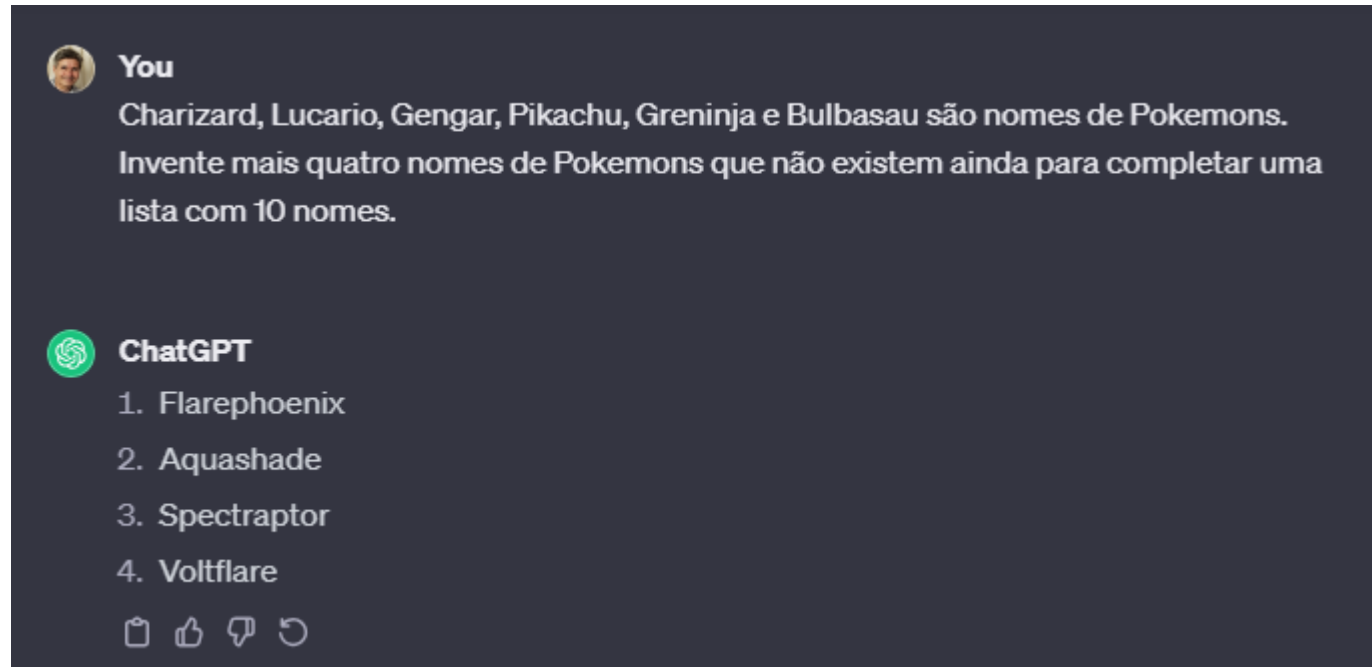


Figura 38



## 2.5. Tokenização

O primeiro passo para treinar um modelo de linguagem é converter as frases da linguagem em sequências de *tokens*. *Um token pode ser um único caractere, um pedaço de uma palavra, uma palavra inteira ou mesmo mais de uma palavra, dependendo do algoritmo utilizado para a tokenização.*

Por exemplo, o algoritmo BPE (*Byte-Pair Encoding*) [66] reúne os pares de caracteres que ocorrem mais frequentemente nas palavras em um *token* único. A sentença "Estude na Universidade Corporativa Módulo (UCM)" tem 6 palavras e 47 caracteres. Ao ser convertida pelo BPE são gerados 14 *tokens* (mostrados na figura em cores diferentes na Figura 39). O teste foi feito no Tokenizer [67].

O total de *tokens* gerados cria um **vocabulário** (em *tokens*). Quanto maior o vocabulário, mas recursos de computação são necessários no treinamento. O modelo GPT 3.5 usado no ChatGPT foi *tokenizado* pelo BPE e treinado com 300 bilhões de *tokens*.

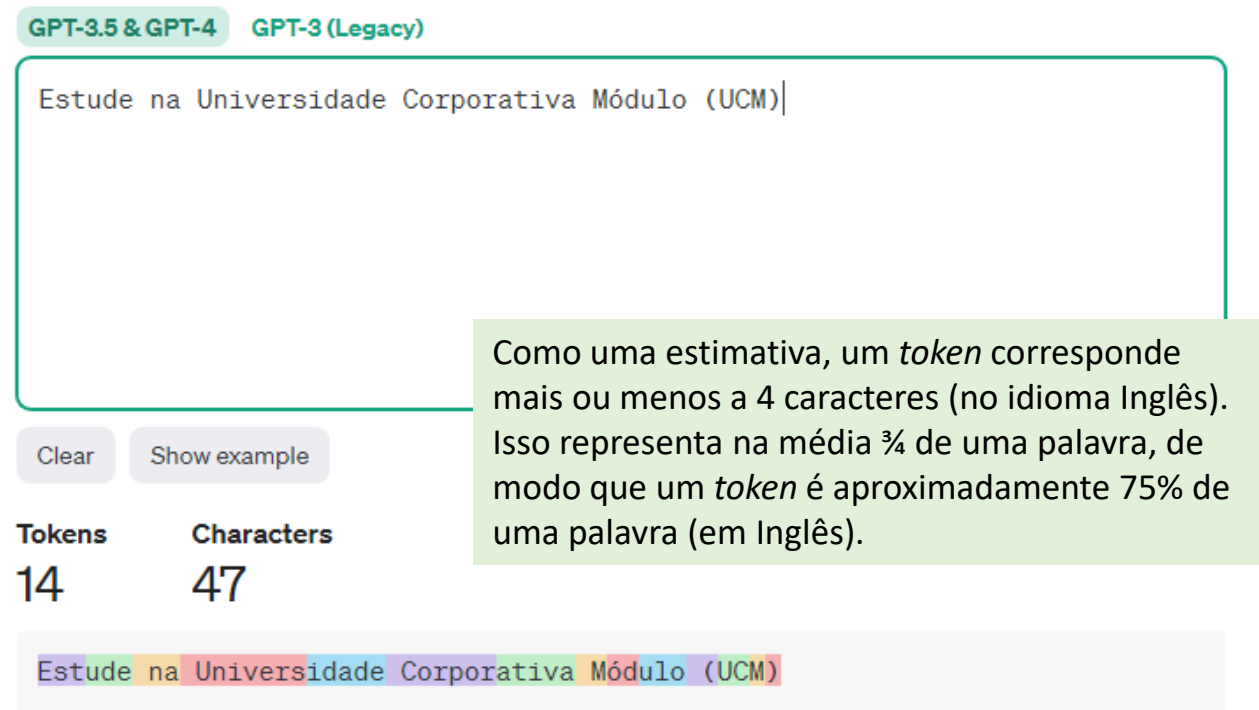


Figura 39 - Fonte: [67]

Podemos confirmar nosso entendimento do que significa um *token* (no contexto do treinamento de modelos de linguagem) perguntando ao próprio ChatGPT. Também podemos perguntar que tipo de *token* foi utilizado no treinamento do modelo que ele (o ChatGPT) está utilizando no momento. O Assistente nos dá uma definição de *token* compatível com a que apresentamos antes, informa que está usando o modelo de linguagem GPT-3.5, e que no treinamento deste modelo foi utilizado o algoritmo BPE (*Byte Pair Encoding*) para a tokenização.

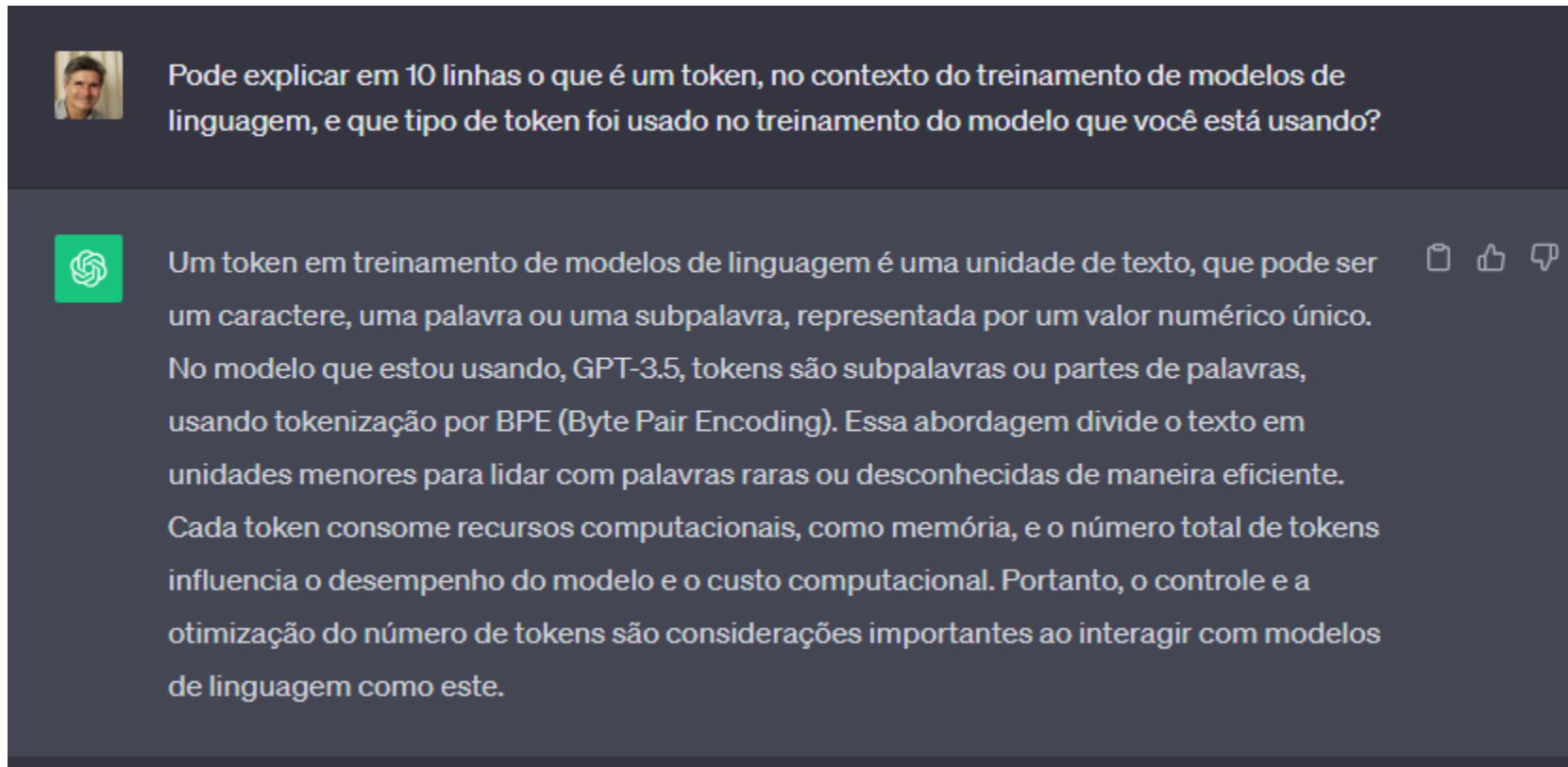


Figura 40

Um problema com esta incrível capacidade generativa dos grandes modelos de linguagem é que eles *mentem*, ou, melhor, erram em suas respostas, podem realmente alucinar em certos casos. No *Prompt* anterior (Figura 40), o ChatGPT completou o texto com uma "reposta" informando que estava usando o modelo de linguagem GPT-3.5, e que no treinamento deste modelo foi utilizado o algoritmo BPE (*Byte Pair Encoding*) para a tokenização. Porém, se você amarrar o ChatGPT em uma cadeira e torturá-lo um pouco, ele bem poderá gerar uma versão diferente (Figura 41).

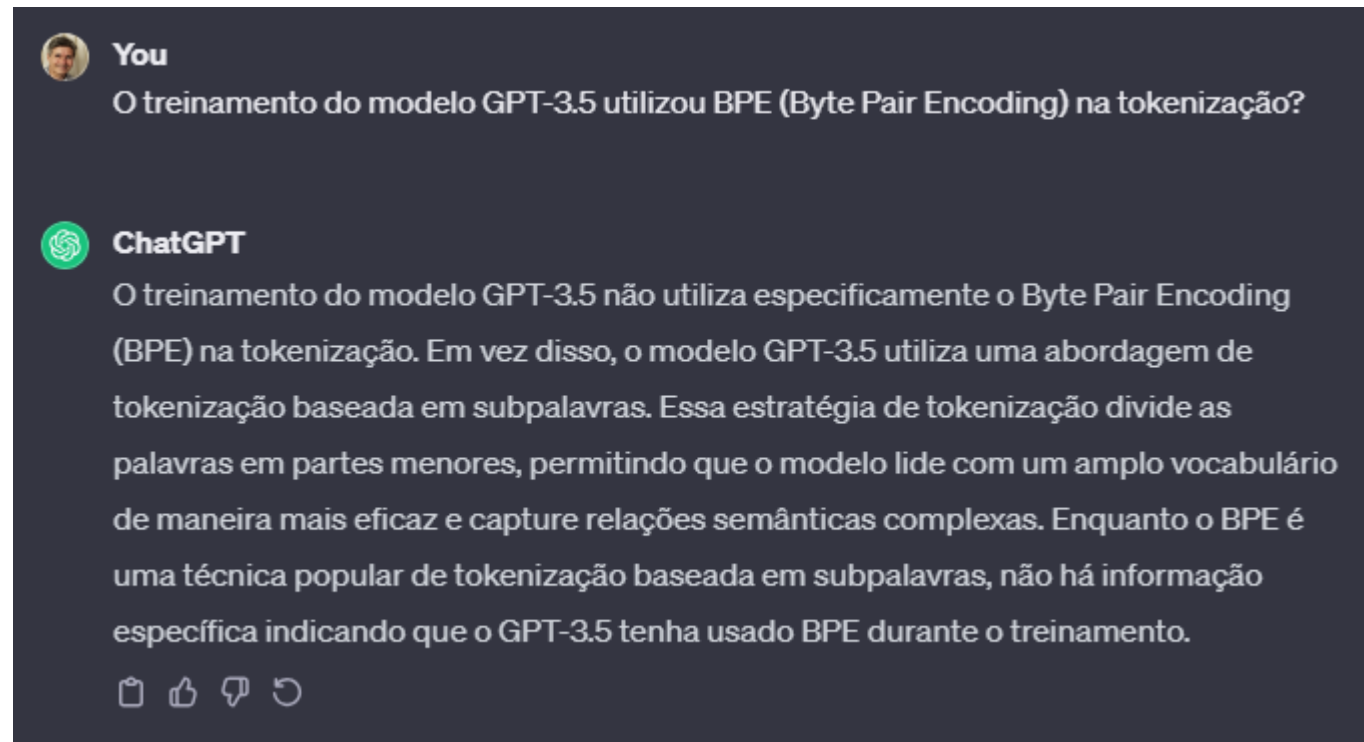


Figura 41

E se você perguntar *novamente*, o escorregadio ChatGPT poderá se contradizer mais uma vez e retornar para a versão anterior. Tenha cuidado!

Nas aplicações importantes do uso da IA Generativa é sempre bom confirmar os conteúdos sintéticos gerados pelos LLMs, que podem ser inventados, inadequados ou simplesmente estar errados no sentido factual.



Figura 42

## 2.6. Word Embeddings

As redes neurais (pelo menos as atuais) trabalham com (ou são alimentadas por) números. Assim, se desejamos utilizá-las para processar textos da linguagem natural, precisamos de uma forma de representar textos através de números.

Aqui surge o importante conceito de "*Word Embeddings*" (algo como "incorporar palavras" em Português).

Tecnicamente, um "*Word Embedding*" [68] é um vetor numérico que representa um *token*. Nos algoritmos de "*Word Embeddings*", as palavras são primeiro convertidas em *tokens*, como vimos na Seção 2.5, e em seguida estes *tokens* são vetorizados (lembre-se que um *token* pode ser um único caractere, ou uma parte de uma palavra, ou uma palavra, ou mesmo uma frase inteira, dependendo do algoritmo de tokenização escolhido).

Assim, o "*Embedding contextual*" é uma técnica que representa *tokens* (vamos imaginar que um *token* é uma palavra para simplificar) através de vetores numéricos, de tal modo que *vetores com valores próximos indiquem palavras com significados próximos*, ou que tendem a ocorrer próximas com maior frequência em um mesmo contexto.

E sendo numéricos (compostos por números reais), estes vetores são perfeitamente adequados para as redes neurais.

É como se cada vetor representando uma palavra existisse em um "espaço de significados" (na verdade, um espaço vetorial de múltiplas dimensões). Utilizando métodos de álgebra linear (similaridade por cosseno - [69]) é possível calcular a distância (ou a similaridade) entre os vetores. Vetores de "*Embeddings*" que estão "próximos" neste espaço vetorial sugerem "significados próximos", considerando similaridade semântica *e também contextual*.

Este conceito é um pouco abstrato, de modo que um exemplo poderá torná-lo mais claro.

Exemplo: A palavra "jacaré" (aligator) *não* é exatamente um sinônimo de "crocodilo" (são répteis diferentes), de modo que estas duas palavras não têm exatamente o mesmo significado. Entretanto, há grande proximidade entre estes conceitos, de modo que é provável que vetores de *Embedding* gerados para representar as palavras "jacaré" e "crocodilo" estejam mais próximos (no espaço vetorial de "significados") do que, por exemplo, vetores que representam as palavras "jacaré" e "camelo".

Podemos inclusive confirmar este entendimento antes de prosseguir com o nosso "coautor" (Figura 43).



You

A palavra "jacaré" (aligator) não é exatamente um sinônimo de "crocodilo" (são répteis diferentes), de modo que estas duas palavras não tem exatamente o mesmo significado, mas há uma grande proximidade entre estes conceitos, de modo que é bem provável que vetores de "Embedding" gerados pelo algoritmo para representar as palavras "jacaré" e "crocodilo" estejam mais próximos (no espaço vetorial de "significados") do que, por exemplo, os vetores que representam as palavras "jacaré" e "camelo". Isso está correto?



ChatGPT

Sim, você está correto. A ideia dos vetores de embedding é capturar a semelhança semântica entre palavras, de modo que palavras semanticamente relacionadas estejam próximas umas das outras no espaço vetorial. No exemplo que você deu, as palavras "jacaré" e "crocodilo" são mais semanticamente próximas do que "jacaré" e "camelo" porque ambas se referem a répteis aquáticos relacionados.

Os algoritmos de embedding, como o Word2Vec, GloVe ou FastText, são treinados em grandes conjuntos de dados linguísticos para aprender representações vetoriais que preservem a semântica das palavras. Portanto, palavras com significados semelhantes terão representações vetoriais mais próximas.

É importante observar que, embora "jacaré" e "crocodilo" sejam semanticamente próximos, eles ainda representam espécies diferentes de répteis e têm diferenças específicas em termos de biologia e habitat. Os embeddings refletem a similaridade contextual e semântica, mas não necessariamente todas as nuances das diferenças específicas entre os conceitos.



Os espaços (matemáticos) onde são criados os vetores de "*Embeddings*" contextuais possuem muitas dimensões, mas podemos ter uma noção observando a Figura 44, que mostra um "espaço de significados" bem simples (adaptado de [43]) contendo apenas duas dimensões, com alguns vetores representando palavras.

Como se constrói um espaço deste tipo?

A ideia é analisar uma quantidade gigantesca de textos (bilhões de palavras obtidas de textos na Internet por exemplo) e verificar os contextos em que as palavras são utilizadas.

É de se esperar uma enorme quantidade de frases que utilizem as palavras "crocodilo" e "jacaré" quase como sinônimos (embora não sejam). Já seria bem menos frequente a presença de "camelo" e "crocodilo" próximas em textos. Assim, os vetores de "*Embeddings*" de "jacaré" e "crocodilo" são mais "próximos", e os de "jacaré" e "camelo" são mais "distantes".

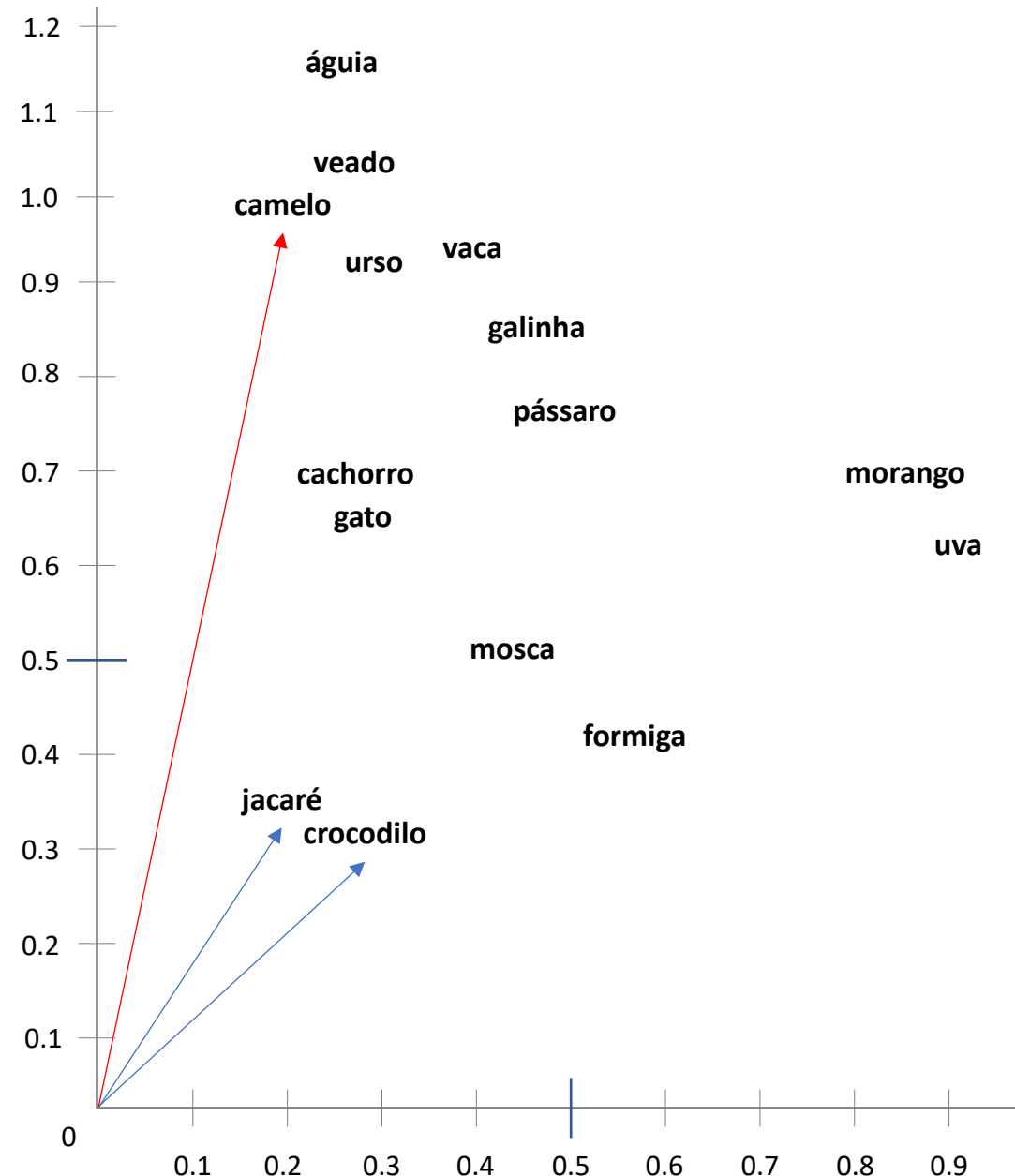


Figura 44 - Fonte: Adaptado de [43] pelo autor

Como explica Stephen Wolfran [43], se observados em seu formato natural ("raw") os vetores de *Embeddings* não são muito informativos para seres humanos. A Figura 45 mostra como exemplos *Embeddings* gerados pelo modelo GPT-2 para as palavras "gato", "cachorro" e "cadeira":

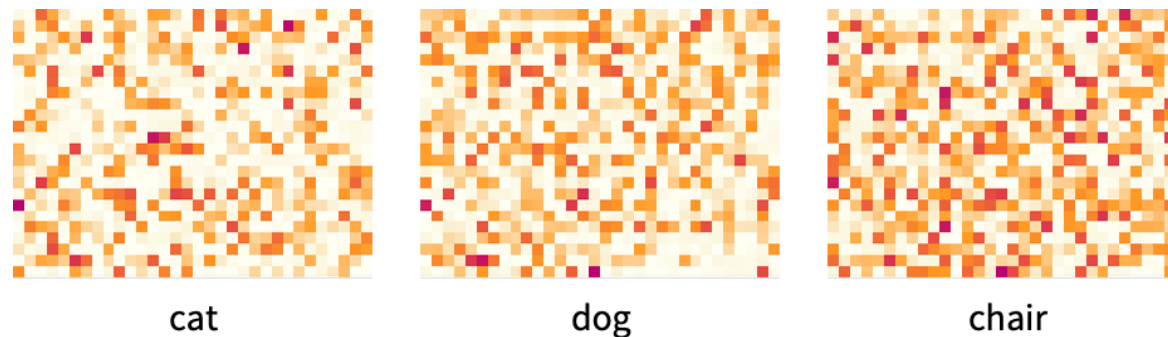


Figura 45 - Fonte: [43]

Se funções matemáticas adequadas forem utilizadas para medir as "distâncias" entre estes vetores, é provável que os vetores para "gato" e "cachorro" (animais domésticos, mamíferos, quatro patas, muito queridos por humanos) estejam mais próximos do que, por exemplo, "cachorro" e "cadeira". Mas o mais importante é que embora sejam ininteligíveis para nós, estes *Embeddings* agora representam palavras (\*) em um formato que é "amigável" para as redes neurais, ou seja, matrizes com números reais.

(\*) Lembrando mais uma vez que na verdade estes vetores não contêm "palavras", mas sim *tokens* (a tokenização ocorre antes da vetorização). O uso de *tokens* torna mais fácil para o ChatGPT lidar com palavras mais raras (que ocorrem com pouca frequência), ou palavras compostas, ou palavras em outros idiomas, ou como diz Stephen Wolfran, "até mesmo inventar novas palavras, para o bem ou para o mal". Como novos nomes para Pokemons.



Refletindo sobre os "*Embeddings*", é surpreendente que uma simples aritmética com vetores consiga capturar relacionamentos de significado (semânticos) ou algum outro tipo de proximidade contextual entre palavras, mas o fato é que na prática este processo funciona muito bem, embora não seja infalível e também possa ter algumas consequências éticas indesejáveis (ver Nota). Por conta disso, este método é utilizado amplamente em diferentes tarefas envolvendo processamento de linguagem natural ou NLP.

No exemplo de [70], dado o desafio “Homem está para rei assim com mulher está para x”,

uma aritmética simples com os vetores que representam estas palavras permite calcular que x=rainha é a melhor resposta. Analogamente, utilizando vetorização (*Word Embeddings*), é possível obter

x = Japão

... como palavra mais provável para completar "Paris está para França, assim como Tokio está para x".

**Nota!** Observar que há considerações éticas envolvendo *bias* (preconceitos, estereótipos) nas associações produzidas por este tipo de técnica. Afinal, homem também pode ser enfermeiro (termo normalmente associado com mulheres), e mulheres também podem ser caminhoneiras (profissão tipicamente masculina). Ver [70].

Um exemplo clássico de algoritmo para vetorização é o Word2Vec [71] lançado em 2013, que utiliza uma rede neural para localizar sinônimos (= palavra diferente, mesmo significado) e *sugerir palavras para completar um texto*, utilizando uma função matemática [69] para calcular a distância (ou similaridade) entre vetores que representam palavras.

Substituindo o Word2Vec, em 2018 surgiram métodos de *Word Embedding* mais avançados (sensíveis ao contexto), como os utilizados nos modelos de linguagem ELMo, o BERT e o GPT. Falaremos sobre eles na Seção 2.12.

## 2.7. Qual é o próximo *token* mais provável na sequência?

Um modelo de linguagem autoregressivo baseado em redes neurais infere uma probabilidade para cada *token* em uma sequência. De forma simplificada, **o propósito do modelo é prever qual é o próximo token da sequência**, ou seja, *qual é o próximo token no vocabulário que tem maior probabilidade P de completar uma sequência prévia de outros tokens*. Neste sentido, os *tokens* são os elementos fundamentais para a geração de texto, como pequenos bloquinhos de lego.



Figura 46

No exemplo seguinte utilizamos palavras em vez de tokens para simplificar. Considerando o contexto (sequência já existente) o modelo calcula probabilidades e assim escolhe que a próxima palavra mais provável para completar a sequência é "voando" (92%). Note que outras palavras como "Júpiter" têm baixíssima probabilidade associada, mas todas as palavras do dicionário são avaliadas.

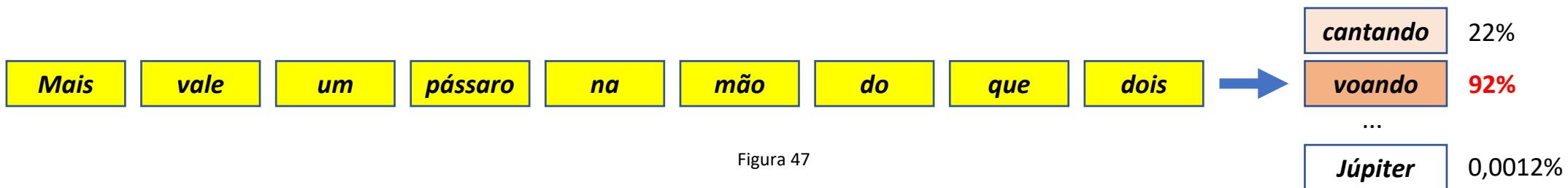


Figura 47

Vejamos o processo anterior com um pouquinho mais de detalhes. Como para computadores é mais fácil trabalhar com números, um aspecto importante é a conversão dos *tokens* em vetores numéricos.

1. *Palavras são convertidas em tokens.* Uma palavra pode gerar mais de um *token* pela tokenização BPE. Porém, outros métodos de tokenização podem ser utilizados. Nos exemplos seguintes vamos assumir que um *token* representa uma palavra inteira.



Figura 48

2. *Tokens são mapeados para números inteiros.* Após as palavras serem convertidas em *tokens*, cada *token* é associado a um índice único. O processo envolve a criação de um **vocabulário** no qual cada palavra (ou subpalavra, dependendo do método de tokenização) é mapeada para um número inteiro (Figura 49). A correspondência palavra-índice é utilizada para converter os *tokens* em sequências contendo apenas seus índices.

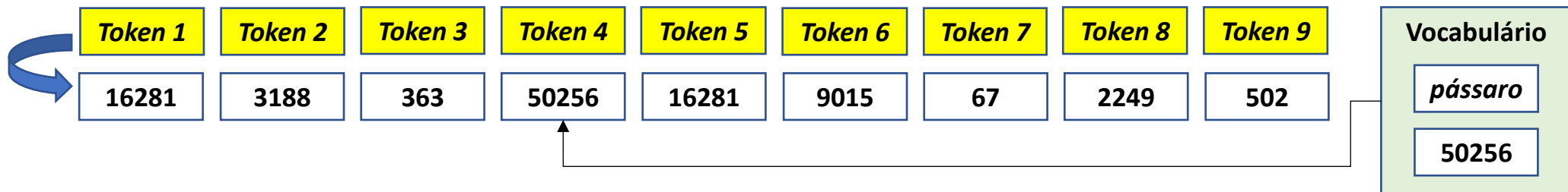


Figura 49

3. *Embedding posicional*. Os números inteiros que identificam os *tokens* são passados para a camada de *Embedding* do ENCODER, um dos componentes da arquitetura da rede neural *Transformer* que treina o modelo. Cada número que representa um *token* específico é associado a um **vetor  $x$  de *Embedding*** (\*), composto por *números reais*.

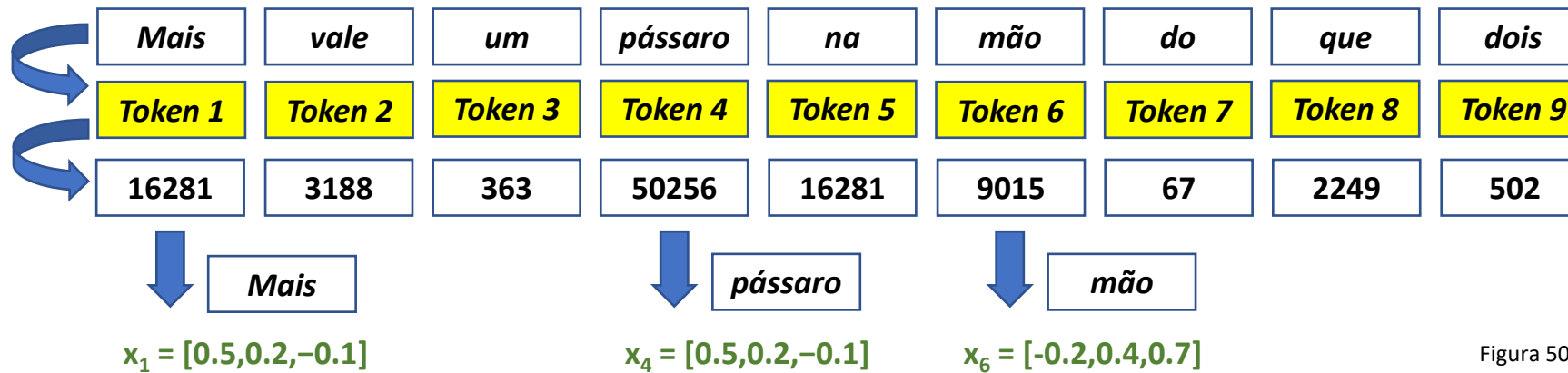


Figura 50

Normalmente, os **vetores  $x_j$  de *Embeddings*** têm muitas dimensões (podem ser centenas), mas para simplificar as palavras "mais", "pássaro" e "mão" foram associadas a vetores de apenas 3 dimensões (por exemplo, o **vetor  $x_1$  de *Embedding*** para a palavra na posição  $j=1$  da sequência ("mais") é  $x_1 = [0.5, 0.2, -0.1]$ ). Estes números representam a posição relativa de cada palavra em um espaço vetorial (o tal "espaço de significados"), o que permite *avaliar as relações semânticas* ou de proximidade contextual entre as palavras que vão emergindo durante o treinamento do modelo. Por exemplo, palavras que compartilham contextos semelhantes em textos como "Saturno" e "anéis" tendem a ter representações vetoriais mais próximas no espaço de *Embedding*.

(\*) Tecnicamente, há ainda um passo - os **vetores  $x_j$  de *Embeddings*** são adicionados a um outro tipo de vetor criado para ajudar o modelo a controlar a posição de cada *token* (palavra) na sentença, gerando os chamados *Embeddings Posicionais*. Estes detalhes não são importantes agora, mas se você estiver curioso, veja o BOX 7.

4. *Treinamento supervisionado*. Após a criação dos **vetores de Embedding Posicional** e a representação das palavras em um espaço vetorial, o treinamento do modelo GPT prossegue. Parte do treinamento é supervisionado por especialistas humanos. No caso, a tarefa que nos interessa é "prever o próximo token mais provável em uma sequência".

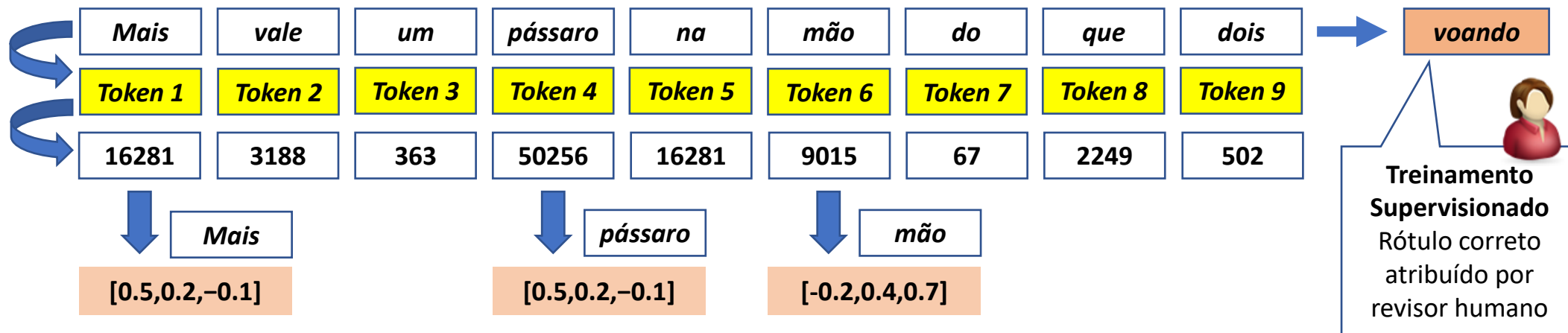


Figura 51

Durante o treinamento supervisionado, o modelo recebe uma entrada parcial da sequência e é orientado a gerar a palavra seguinte. Uma **função de perda** mede a diferença entre as previsões do modelo e as saídas reais (rótulos fornecidos por revisores humanos). O objetivo é minimizar este erro ou perda.

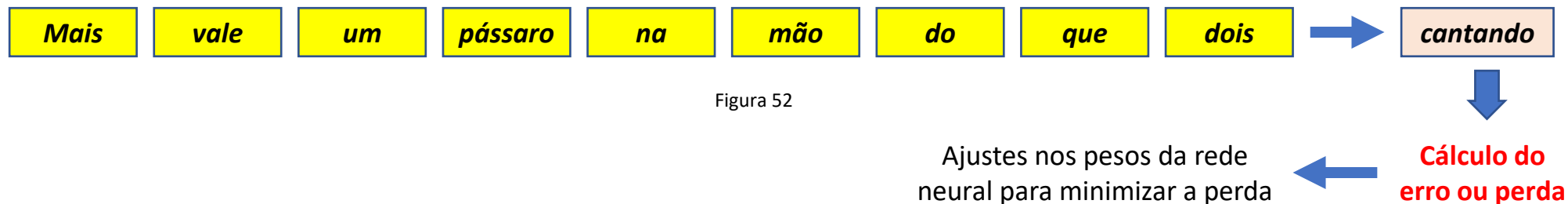


Figura 52

No caso da tarefa "predizer o próximo *token*" uma função de perda bastante utilizada é a perda por entropia cruzada (*Cross-entropy Loss*) [72]. A entropia cruzada é eficaz para medir a discrepância entre a distribuição de probabilidades predita pelo modelo e a distribuição de probabilidades real dos rótulos atribuídos pelos revisores humanos em tarefas de classificação.

Seguindo o exemplo da Figura 52, digamos que a palavra gerada pelo modelo para completar a sentença seja "cantando", quando na verdade a palavra esperada é "voando". A diferença entre a previsão e a palavra real resulta em uma medida de erro ou perda. Este erro é retropropagado ("propagado para trás") através das diversas camadas da rede neural, e os pesos e parâmetros do modelo são reajustados visando minimizar essa perda e melhorar o desempenho do modelo durante o treinamento. Esse processo é repetido diversas vezes, com grandes conjuntos de dados, até que se alcance uma performance (um nível de erros) aceitável. Como veremos no Capítulo 3, a parte inicial do treinamento de um modelo GPT é não supervisionada, mas também há etapas onde o treinamento é supervisionado (por especialistas humanos).

Uma vez treinado, o modelo será capaz de predizer o próximo *token* (próxima palavra com maior probabilidade de completar a sentença) com boa precisão na maioria dos casos, bem como realizar outras tarefas de linguagem, como tradução de textos, ou análise de sentimentos, dependendo de para que finalidade o modelo foi treinado.

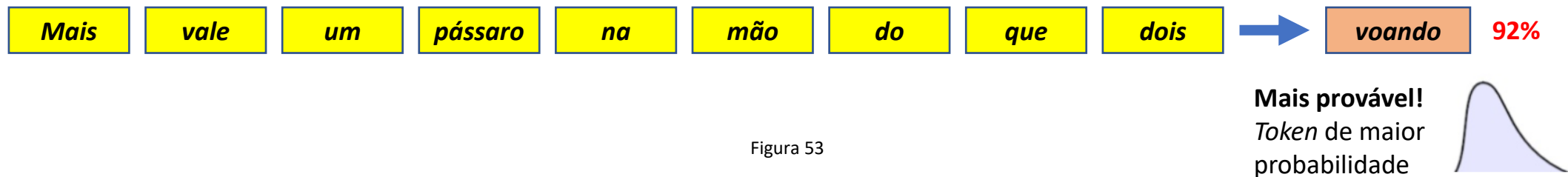


Figura 53

E assim já temos pelo menos alguma intuição de como funciona o modelo GTP 3-5 que está "por trás dos panos" completando textos, um *token* de cada vez, quando passamos instruções para o ChatGPT.

A propósito, a sentença que desejamos completar é chamada *Prompt*.

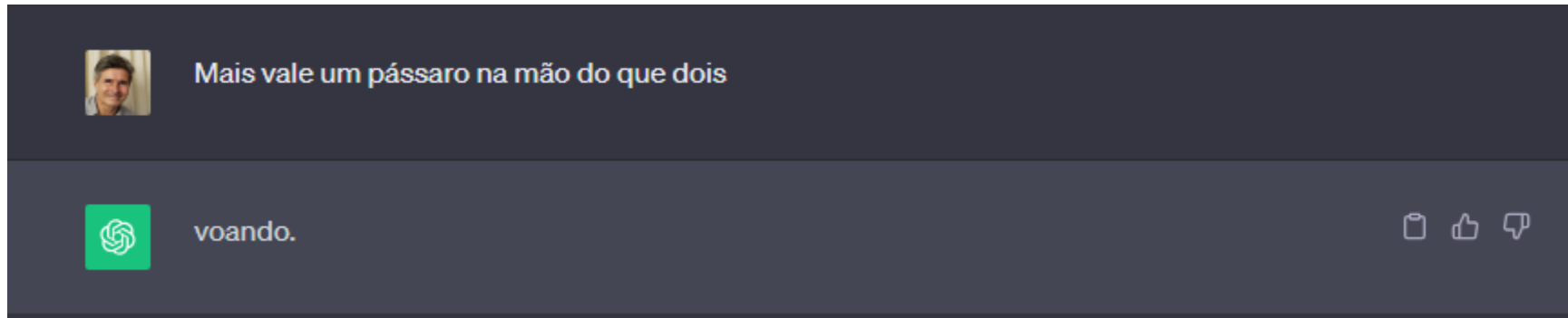


Figura 54

É claro que o que discutimos aqui é uma simplificação. Podemos nos aproximar um pouquinho mais do que realmente acontece quando utilizamos o ChatGPT aprendendo sobre a **amostragem softmax**.

## 2.8. Amostragem softmax

Vamos elaborar um pouco mais sobre como se descobre o *token* mais provável para completar a sentença "eu vou retirar dinheiro no"....? (ou qualquer outra que o ChatGPT precise completar).

O próximo *token* (mais provável) é obtido por *sampling* de uma distribuição de probabilidades, da seguinte forma:

**Passo 1** - Primeiro, é necessário calcular a distribuição de probabilidades para cada próximo *token* (*considerando todo o vocabulário*), *com base no contexto fornecido pelos tokens anteriores na sequência* (uma probabilidade condicional, como veremos mais adiante).

Isso geralmente é feito por um modelo de linguagem já treinado baseado em redes neurais *Transformer*, como os modelos GPT-3.5 ou GTP-4 no caso do ChatGPT. Aqui, estamos falando de *calcular - em tempo real - probabilidades para mais de 50.000 tokens* (o vocabulário utilizado no treinamento do modelo GPT-3 tinha 50.257 *tokens* diferentes).

Tecnicamente, o que o modelo de linguagem gera não é ainda uma distribuição de probabilidades, mas um **vetor de logits** contendo pontuações numéricas que refletem as "preferências" do modelo pelos diferentes *tokens*. Estas pontuações são números reais que podem ser positivos (*tokens* preferidos pelo modelo), nulos (indiferentes) ou negativos (*tokens* que o modelo não prefere).

Se o vetor de *logits* pode conter valores negativos, ele de fato não conter "probabilidades", já que não existem probabilidades negativas. A verdadeira distribuição de probabilidades é gerada pela função softmax [73], que faz a normalização das pontuações do vetor de *logits* e as converte em probabilidades. A forma da distribuição gerada pela softmax é afetada pelo hiperparâmetro **temperatura** (mais detalhes no BOX 3).



**Passo 2** - As probabilidades geradas pela função softmax são *ordenadas da maior para a menor* em um vetor chamado *vetor de probabilidades* ou *vetor de distribuição de probabilidades*, cujos valores (positivos ou nulos) representam uma probabilidade (como 0.031 ou 3,1%) para cada um dos *tokens* no dicionário. Os valores são normalizados, de modo que a soma de todos os valores neste vetor é igual a 1 ou 100%.

**Passo 3** - Já de posse de um vetor ordenado contendo uma ***distribuição de probabilidades***, o próximo passo é fazer uma amostragem nesta distribuição, obtendo um *token* do vocabulário. Aqui é utilizada uma técnica chamada *amostragem softmax* [73, 74].

A amostragem softmax faz a seleção de *tokens* de forma que seja possível que outros *tokens* (que *não são* os de maior probabilidade) também possam ser escolhidos ou amostrados, gerando assim maior diversidade nas respostas produzidas pelo modelo.

Vejamos um exemplo para tornar o processo mais claro.

Seja a seguinte distribuição de probabilidades para os próximos *tokens* mais prováveis para completar a frase

"eu vou retirar dinheiro no"....:

- > "caixa" (probabilidade 0.4)
- > "banco" (probabilidade 0.3)
- > "supermercado" (probabilidade 0.2)
- > "restaurante" (probabilidade 0.1).

...

(e mais milhares de outras probabilidades estimadas para outros 50.000 tokens possíveis do vocabulário).

Neste caso, teríamos um longo vetor com os valores das probabilidades para todo o vocabulário.

"eu vou retirar dinheiro no"...

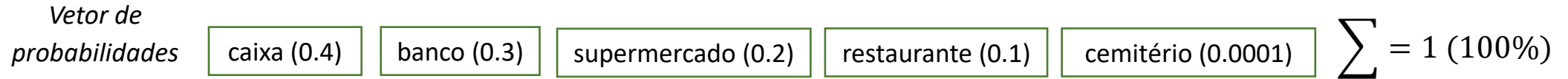


Figura 55

Note que no vetor as probabilidades estão *ordenadas da maior para a menor*, e somam 1 (100%) (considerando todo o vocabulário). *Tokens* com probabilidade muito baixa foram ignorados (ou teríamos que desenhar 50.000 caixinhas). Para fazer a amostragem, a função softmax gera um número aleatório entre 0 e 1, digamos, 0.65. Então, o vetor com as probabilidades é percorrido e as probabilidades vão sendo acumuladas até que a soma acumulada seja maior que 0.65. No caso do exemplo, "banco" seria o *token* amostrado, pois a soma acumulada após "caixa" e "banco" seria  $0.7 > 0.65$ .

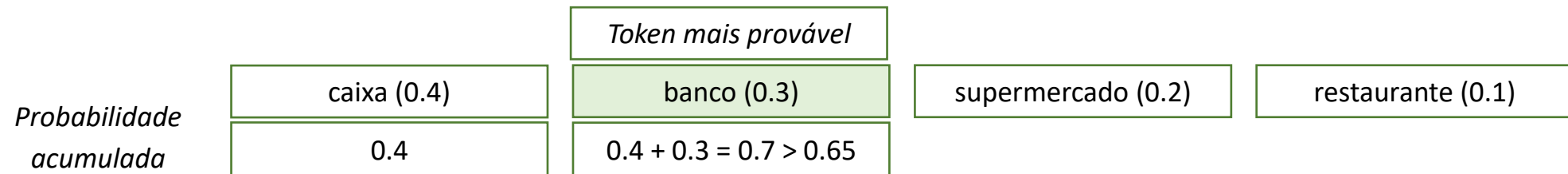


Figura 56

A acumulação de probabilidades após "caixa" é 0.4, e após "banco" é 0.7. Como 0.7 é maior que 0.65, o *token* "banco" é escolhido como o *token* amostrado pela softmax. *Note que neste exemplo o token de maior probabilidade (caixa, 40%) não foi o token escolhido.* Em vez disso, a softmax amostrou o segundo *token* do vetor (banco, 30%). É desta forma que a softmax ajuda a introduzir maior diversidade nas respostas geradas pelo modelo.

## Revisando:

1. O modelo de linguagem já treinado (por uma rede neural *Transformer*) é utilizado para gerar um vetor de *logits*, uma distribuição de valores numéricos que representam as saídas brutas ou não normalizadas do modelo *antes* da aplicação da função softmax. Cada elemento do vetor de *logits* é um número real que pode ser positivo, nulo ou negativo. *Logits* positivos e mais altos indicam maior "preferência" do modelo por determinada categoria ou classe.
2. Quando aplicamos a função softmax ao vetor de *logits*, ela transforma esses valores em uma distribuição de probabilidades. Ou seja, é gerado um novo vetor, desta vez composto por probabilidades (números entre 0 e 1) para cada *token* do dicionário, sendo a soma de todos os elementos igual a 1 ou 100%.
3. O parâmetro "temperatura" que faz parte da função softmax afeta a forma da distribuição gerada (BOX 3). Um valor mais alto de temperatura suaviza a distribuição gerada, tornando-a mais uniforme, enquanto um valor mais baixo aguça a distribuição, destacando as diferenças entre as probabilidades.
4. No vetor de probabilidades cada elemento representa a probabilidade associada a cada um dos *tokens* no dicionário (por exemplo, 50.000 *tokens*) dado o contexto fornecido. Os valores de probabilidade são ordenados de forma decrescente no vetor, facilitando a interpretação e a tomada de decisões do modelo.
5. Vale ressaltar que o modelo considera a *sequência completa de tokens anteriores* ao calcular a distribuição de probabilidades para os *tokens* do dicionário. Ou seja, trata-se de uma *probabilidade condicional*, onde o modelo leva em conta o contexto global da conversa ou frase (o contexto global é o conteúdo que passamos em cada *Prompt* para o ChatGPT, além dos conteúdos que ele próprio gera em cada interação).
6. Gerado o vetor de probabilidades, é iniciado o processo de *sampling* de *tokens* por amostragem estocástica (técnica que envolve a geração de várias possíveis respostas e, em seguida, a escolha da resposta mais provável com base em um conjunto de critérios, utilizada nos modelos de linguagem baseados em *Transformer* como o GPT-3).

7. Na amostragem, um valor aleatório  $r$  é gerado, onde  $r$  está no intervalo no intervalo aberto  $(0,1)$  - o que significa que não pode ser exatamente zero nem exatamente um. Este valor aleatório é usado para decidir qual *token* amostrar. O vetor de probabilidades é "percorrido" até que o valor aleatório seja atingido. Isso identifica o *token* que deve ser amostrado. Se o valor aleatório for *menor* que a probabilidade do primeiro *token* no vetor de probabilidades, então o primeiro *token* será escolhido. Por outro lado, se o valor aleatório for muito alto (por exemplo, 0.88 ou 88%), vários *tokens* do vetor poderão ser percorridos até que as suas probabilidades acumuladas alcancem o valor aleatório. Neste caso, poderá ser escolhido por exemplo o segundo, ou o terceiro *token* mais provável, e não sempre o *token* o MAIS provável. Afinal, se fosse para escolher sempre o *token* mais provável, seria suficiente selecionar sempre o *primeiro token* do vetor, que é o de maior probabilidade (ver Nota).

Tanto o parâmetro de "temperatura" quanto a aleatoriedade na amostragem da função softmax tornam possível de formas diferentes que seja escolhido um *token* que *não* seja o de probabilidade mais alta para completar a sentença. A temperatura afeta como as probabilidades são distribuídas *antes* da amostragem softmax (BOX 3), enquanto o valor aleatório entre 0 e 1 introduz aleatoriedade na escolha real do *token* com base nessas probabilidades. Ambos os elementos são importantes para ajustar a diversidade e a determinismo na geração de texto.

**Nota!** A possibilidade de selecionar (por amostragem) um *token* que *não* seja o de maior probabilidade pode comprometer um pouco a precisão das respostas, que serão menos "determinísticas", mas ajuda a gerar respostas mais diversificadas e naturais para os seres humanos. Esta é a abordagem utilizada nos modelos de linguagem treinados por redes *Transformer*, como os utilizados pelo ChatGPT. A alternativa seria escolher sempre o *token* de maior probabilidade (através de uma função *argmax*) no vetor de probabilidades. Esta abordagem é chamada "Escolha Gulosa" (*Greedy Sampling*), e não completa textos de modo interessante pois não permite muita variabilidade nas respostas, já que o primeiro *token* do vetor de probabilidades seria sempre o escolhido se o mesmo *Prompt* fosse passado diversas vezes.

### BOX 3. A função softmax - Transformando *logits* em probabilidades

A **temperatura** [75] é um hiperparâmetro que multiplica as *logits* (resultados da última camada linear da rede neural *antes* da aplicação da função softmax de amostragem). A função softmax (regressão logística multinomial) converte (normaliza) o vetor de *logits* com todos os seus elementos (o vocabulário inteiro) para um **vetor de probabilidades** (ou distribuição de probabilidades, de modo que a soma dos elementos do vetor de probabilidades é igual a 1 ou 100%. O valor escolhido para a temperatura afeta a *forma* da distribuição de probabilidades que será gerada:

- Valores maiores de temperatura (por exemplo,  $> 1$ ) suavizam a distribuição de probabilidades, tornando-a mais uniforme. Isso significa que *tokens menos prováveis terão uma chance maior de serem escolhidos*, e assim as respostas geradas pelo modelo serão, digamos, mais "criativas" (embora talvez menos precisas).
- Valores menores de temperatura (por exemplo,  $< 1$ ) aguçam a distribuição de probabilidades, destacando as diferenças entre as probabilidades dos *tokens*. Isso torna a amostragem mais determinística - *os tokens com maior probabilidade provavelmente serão os escolhidos*, e as respostas do modelo serão menos criativas.

A fórmula geral para a função softmax com temperatura é:

$$\text{Softmax}_i(x) = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}}$$

Onde

- $x$  é o vetor de logits antes da aplicação da softmax.
- $x_i$  é um *token* na  $i$ -ésima posição no vetor.
- $x_j$  é usado na soma no denominador (o somatório que é feito sobre todos os elementos do vetor de *logits*).
- $T$  é o parâmetro de temperatura.
- $e$  é o número de Euler (aproximadamente 2.71828)

Figura 57 - A função softmax converte os *logits* em uma distribuição de probabilidades, e o uso do parâmetro de temperatura ( $T$ ) introduz uma flexibilidade adicional. *Esse vetor tem o mesmo tamanho que o vetor de logits, e cada elemento no vetor de probabilidades representa a probabilidade associada a um token específico no vocabulário.*

- Quando  $T$  é alto, as diferenças entre os *logits* são suavizadas, resultando em probabilidades mais uniformes.
- Quando  $T$  é baixo, as diferenças entre os *logits* são amplificadas, levando a probabilidades mais pontiagudas.

Na fórmula, a soma no denominador que normaliza os resultados é realizada sobre todos os *tokens* do vocabulário (cada posição  $x_i$  no vetor de *logits* corresponde a um *token* específico no vocabulário).

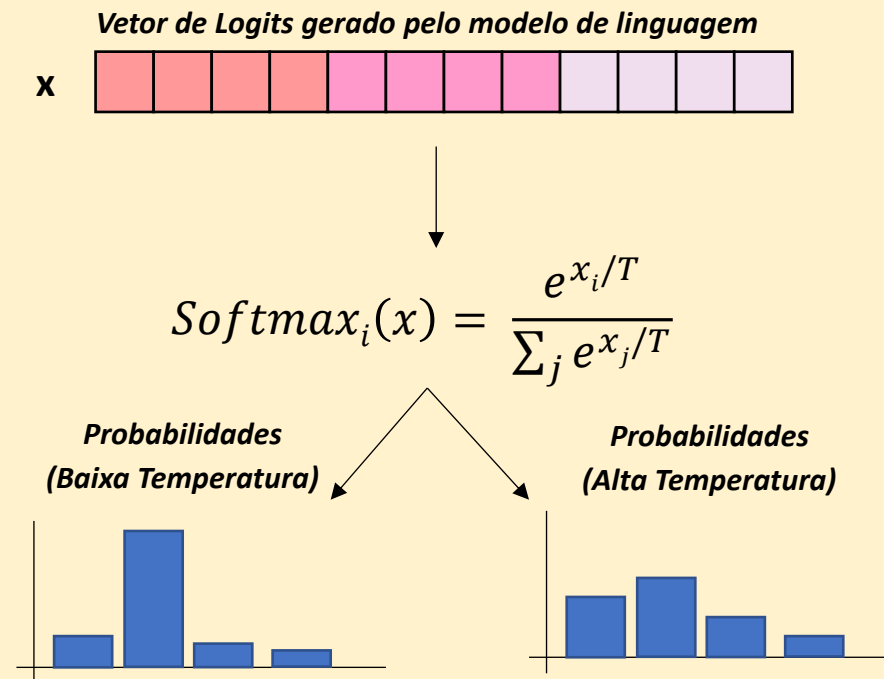


Figura 57 - Fonte: Adaptado de [75] pelo autor

## 2.9. Modelo de Linguagem autoregressivo

Agora que já temos uma noção intuitiva, podemos dar uma definição menos informal para **Modelo de Linguagem** (mais precisamente, dos modelos de linguagem *autoregressivos* utilizados na geração de textos), adaptada de [44].

Seja  $\mathbf{T}$  uma sequência formada por  $n$  *tokens*  $\mathbf{T} = \{t_1, t_2, t_3, \dots, t_n\}$ .

Um Modelo de Linguagem assume que "existe uma probabilidade  $P$  associada com a ocorrência de uma sequência de *tokens*  $t_{1:n}$ , representada por  $P(t_{1:n})$ , onde  $t_{1:n}$  (lê-se " $t$  de 1 até  $n$ "), sendo que  $n$  representa a posição do último *token* na sequência considerada.

A seguinte fórmula (regra da cadeia) nos permite calcular a probabilidade  $P$  associada com a sequência iniciada pelo *token*  $t_1$  e completada pelo *token*  $t_n$ :

$$P(t_{1:n}) = P(t_1)P(t_2|t_1)P(t_3|t_{1:2})P(t_4|t_{1:3}) \dots P(t_n|t_{1:n-1}) \quad [2.9]$$

Note que se trata de uma *probabilidade condicional*, isto é, "a probabilidade de um evento ocorrer, considerando que um outro evento já ocorreu" (a notação  $P(A|B)$  significa "a probabilidade do evento  $A$  ocorrer, considerando que o evento  $B$  já ocorreu").

Assim, por exemplo, o termo  $P(t_4|t_{1:3})$  na fórmula [2.9] significa "a probabilidade do *token*  $t_4$  ocorrer (ser o próximo da sequência), considerando que os *tokens*  $t_1$  até  $t_3$  já ocorreram".

Note também que a fórmula [2.9] para cálculo de  $P(\mathbf{t}_{1:n})$  envolve a *multiplicação de várias probabilidades*.

Caso algum destes valores seja muito pequeno há o risco de serem arredondados para zero durante os cálculos [44]. Para evitar isso, em vez de multiplicar, em geral se calcula o *log* de cada termo se faz uma soma, o que é uma operação equivalente e evita o problema de que valores muito pequenos no somatório "zerem" todo o resultado.

$$\log P(\mathbf{t}_{1:n}) = \log P(\mathbf{t}_1) + \log P(\mathbf{t}_2 | \mathbf{t}_1) + \log P(\mathbf{t}_3 | \mathbf{t}_{1:2}) + \log P(\mathbf{t}_4 | \mathbf{t}_{1:3}) + \dots + \log P(\mathbf{t}_n | \mathbf{t}_{1:n-1})$$

Enquanto o modelo é treinado, ele compreende a estrutura da linguagem - após ser alimentado com uma quantidade massiva de textos, vai aos poucos aprendendo quais palavras ocorrem com maior frequência em proximidade de outras, em diferentes contextos (como decorrência das regras gramaticais de cada idioma).

Muito bem, temos então uma definição formal de modelo de linguagem autoregressivo, e vimos que a probabilidade condicional é parte importante nesta definição.

Como tudo isso se relaciona com a amostragem de *tokens* feita pela função **softmax**, que vimos na Seção anterior?

Como vimos, a função softmax é comumente usada em modelos de linguagem autoregressivos para converter um vetor de saída (vetor de *logits*) em um vetor de probabilidades. Assim, as pontuações de "preferência" geradas pelo modelo para todos os *tokens* no vocabulário são transformadas em probabilidades, cuja soma é igual a 1.

Vimos também que amostragem softmax é uma técnica usada para escolher um *token* dentre todos os *tokens* do vocabulário com base nas probabilidades. A ideia é que *tokens* com maiores probabilidades tenham mais chances de serem escolhidos, mas *tokens* com probabilidades menores também tenham alguma chance de serem escolhidos.



Ao utilizar a amostragem softmax para escolher o próximo *token*, o modelo leva em consideração a probabilidade condicional. *Os tokens anteriores na sequência influenciam diretamente as probabilidades atribuídas a todos os tokens possíveis, afetando a escolha final.*

O fato dos modelos de linguagem autoregressivos calcularem a probabilidade condicional do próximo *token* em relação ao contexto é crucial. Em um modelo autoregressivo, como o *Transformer*, por exemplo, o contexto é representado pelos *tokens* gerados anteriormente na sequência. A probabilidade de um *token* específico (digamos, *token "t"*) ocorrer é condicional aos *tokens* anteriores na sequência.

Isso permite que o modelo gere sequências que fazem sentido em um contexto mais amplo da linguagem, mantendo a coerência e a fluidez nas gerações de texto.

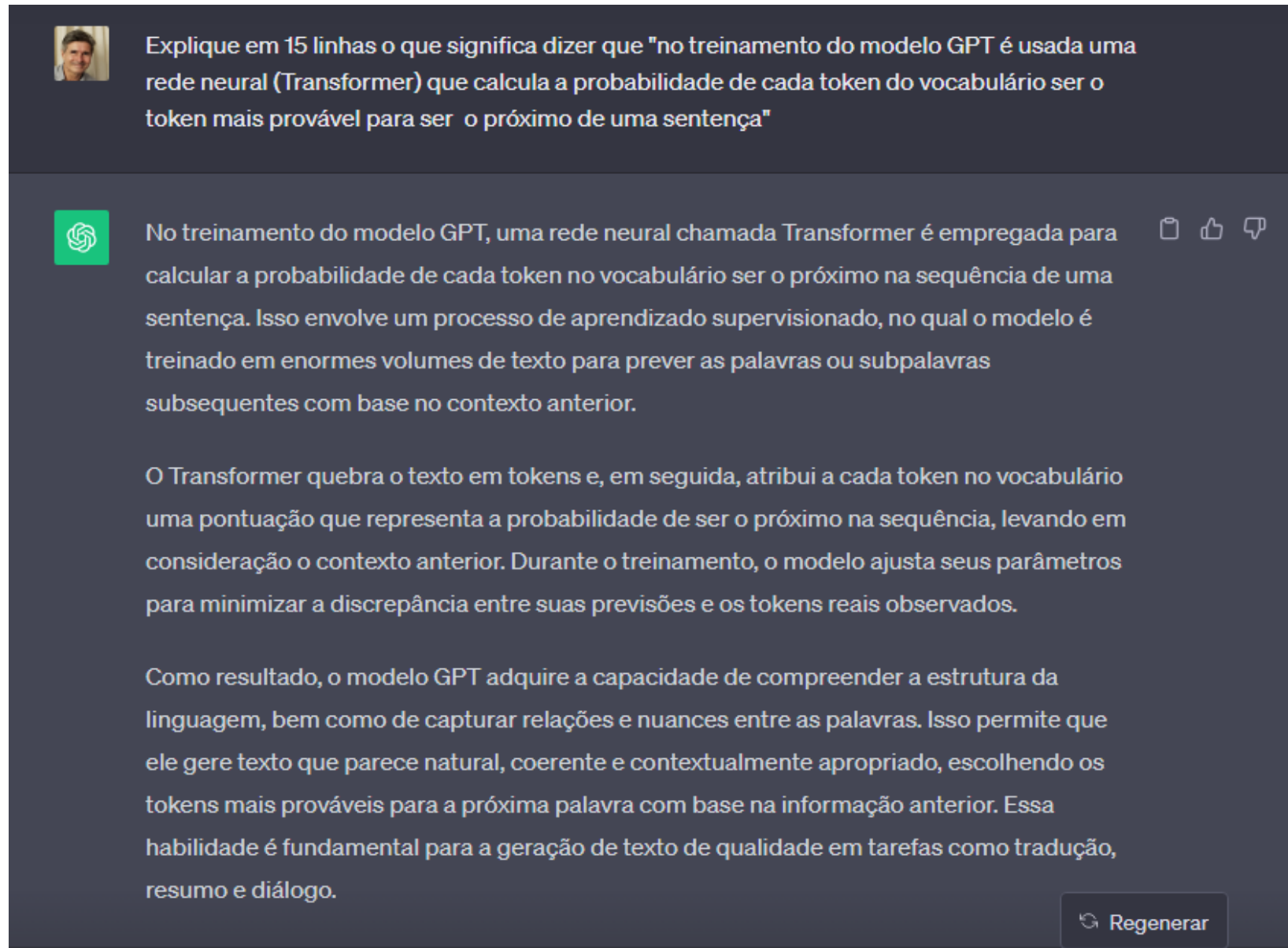
Em resumo, a amostragem softmax é uma forma de incorporar a *natureza autoregressiva dos modelos de linguagem*, onde a probabilidade do próximo *token* é influenciada pelo contexto fornecido pelos *tokens* anteriores na sequência. Isso contribui para a capacidade do modelo de gerar texto mais coeso e contextualmente mais relevante.

Embora cometa erros, como sabemos, na maior parte dos casos, o ChatGPT consegue "gerar" linguagem natural com bastante sucesso, ou seja, as escolhas feitas pelo modelo das próximas palavras (*tokens*) para completar o texto em geral fazem sentido. Em certos casos o ChatGPT consegue conversar tão naturalmente que é difícil identificar se você está conversando com uma inteligência artificial (como é o caso) ou com um outro ser humano.

De fato, os LLMs mais poderosos já passaram no *Teste de Turing* [76, 77].

Para reforçar o que foi dito, vamos pedir ao ChatGPT para nos explicar o que significa dizer...

"no treinamento do modelo GPT é usada uma rede neural (*Transformer*) que calcula a probabilidade de cada *token* do vocabulário tem maior probabilidade de ser o próximo de uma sentença".



Explique em 15 linhas o que significa dizer que "no treinamento do modelo GPT é usada uma rede neural (*Transformer*) que calcula a probabilidade de cada token do vocabulário ser o token mais provável para ser o próximo de uma sentença"

No treinamento do modelo GPT, uma rede neural chamada *Transformer* é empregada para calcular a probabilidade de cada token no vocabulário ser o próximo na sequência de uma sentença. Isso envolve um processo de aprendizado supervisionado, no qual o modelo é treinado em enormes volumes de texto para prever as palavras ou subpalavras subsequentes com base no contexto anterior.

O *Transformer* quebra o texto em tokens e, em seguida, atribui a cada token no vocabulário uma pontuação que representa a probabilidade de ser o próximo na sequência, levando em consideração o contexto anterior. Durante o treinamento, o modelo ajusta seus parâmetros para minimizar a discrepância entre suas previsões e os tokens reais observados.

Como resultado, o modelo GPT adquire a capacidade de compreender a estrutura da linguagem, bem como de capturar relações e nuances entre as palavras. Isso permite que ele gere texto que parece natural, coerente e contextualmente apropriado, escolhendo os tokens mais prováveis para a próxima palavra com base na informação anterior. Essa habilidade é fundamental para a geração de texto de qualidade em tarefas como tradução, resumo e diálogo.

Regenerar

Figura 58

## 2.10. De onde vem tanto texto?

Em muitos casos, percebemos que a resposta fornecida pelo ChatGPT *contém bem mais informação do que esperávamos*, como no exemplo da Figura 59.

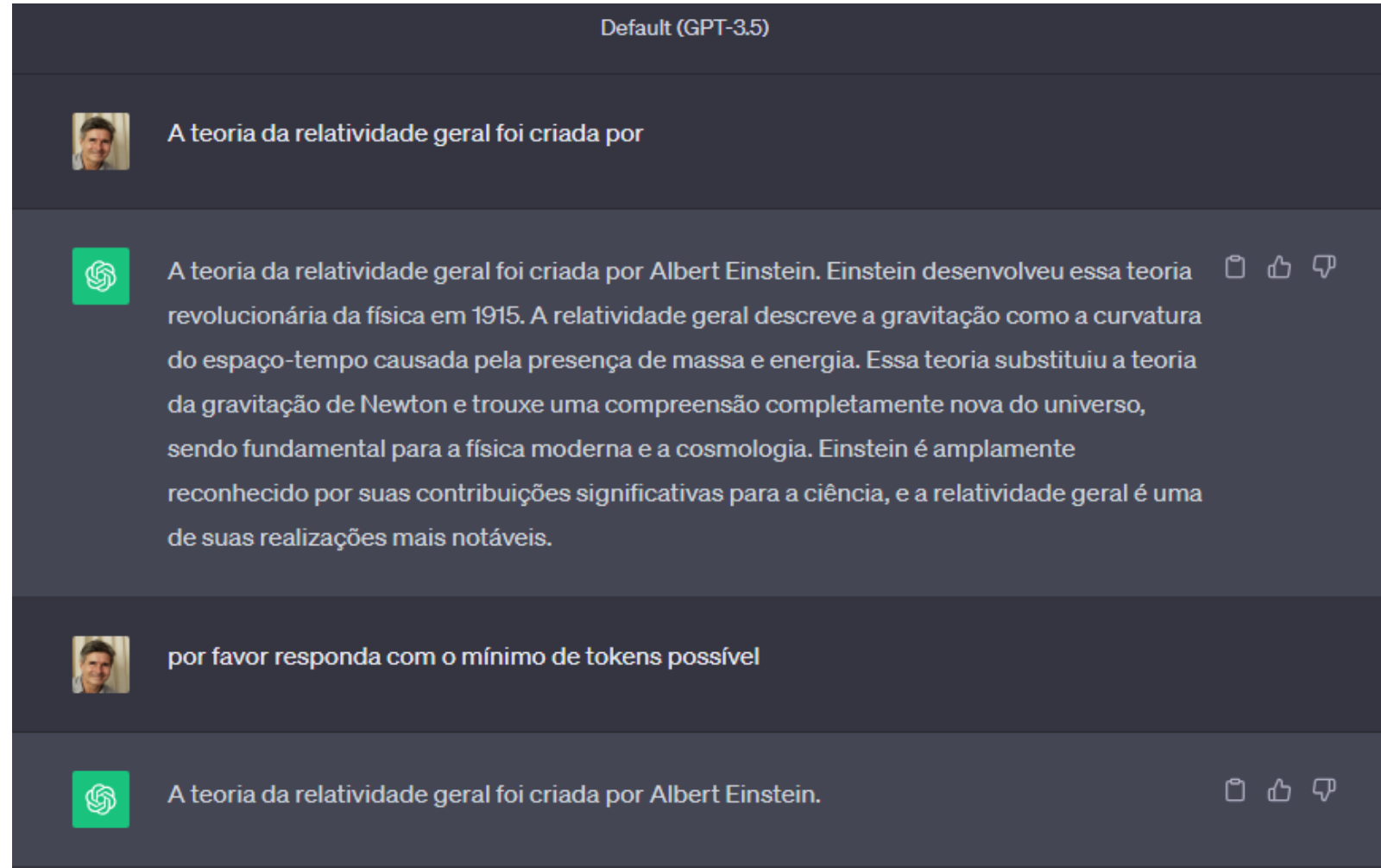


Figura 59

Pelo que foi dito até o momento sobre o mecanismo de treinamento de modelos de linguagem, era de se esperar que o modelo recebesse uma sequência (vetor) com vários *tokens* (contexto), escolhesse o *token* de maior probabilidade para ser o próximo da sequência (ou algum outro também provável), e retornasse apenas ESTE *token*...

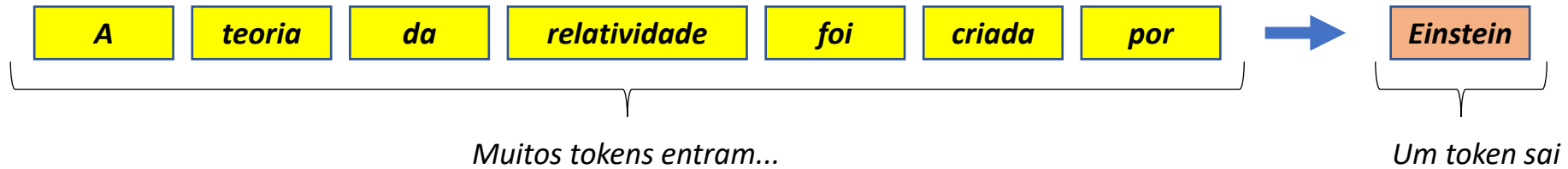


Figura 60

... mas em geral não é isso que ocorre quando usamos o ChatGPT. Mesmo quando fazemos um *Prompt* que pode ser respondido com um ou dois *tokens*, o ChatGPT *continua acrescentando mais texto*. No exemplo da página anterior, passamos o *Prompt* "A teoria da relatividade foi escrita por..." e a resposta poderia ser apenas "Einstein", ou "Albert Einstein", mas o ChatGPT deu uma resposta correta, porém com várias linhas de conteúdo adicional.

De fato, a amostragem softmax estocástica poderia teoricamente escolher apenas um *token* em cada geração de texto. No entanto, na prática, os modelos de linguagem autoregressivos geralmente geram mais de um *token* em cada passo. A quantidade de *tokens* gerados depende da implementação. Modelos autoregressivos geralmente geram múltiplos ramos na geração de texto, e diferentes sequências podem ser aproveitadas na resposta. O modelo pode ter critérios internos para decidir quando parar de gerar *tokens*, como alcançar um *token* de fim de sequência ou atender a algum critério de confiança ou probabilidade mínima. Observe que há um limite para a resposta. O modelo GPT-3.5 por exemplo tem um comprimento máximo da sequência que pode ser gerada em uma única chamada. Se o comprimento máximo for atingido antes de o modelo gerar uma resposta completa, a sequência será truncada ou cortada.

## 2.11. Diversidade nas respostas

A rede neural *Transformer* retorna um vetor no qual cada entrada expressa a probabilidade condicional de um *token* ser escolhido como o mais provável para completar a sentença, considerando os *tokens* que vieram antes (contexto). Portanto, não se trata de um modelo determinístico e nem sempre um mesmo *token* será sugerido para completar uma mesma sequência anterior. Esta aleatoriedade, em geral, é boa. Se o ChatGPT respondesse sempre da mesma forma, a "diversidade" no conteúdo gerado seria baixa (o modelo seria "pouco criativo"). Isso tornaria as respostas menos naturais, pois há muitas palavras que podem fazer sentido para completar frases. Por exemplo, a sequência "A coisa mais preciosa da vida é o...." pode ser completada de forma sensata de diferentes formas. Uma delas seria a seguinte...



Figura 61

Mas o ChatGPT pode completar a sequência de outra forma, cuja compreensão é igualmente aceitável.

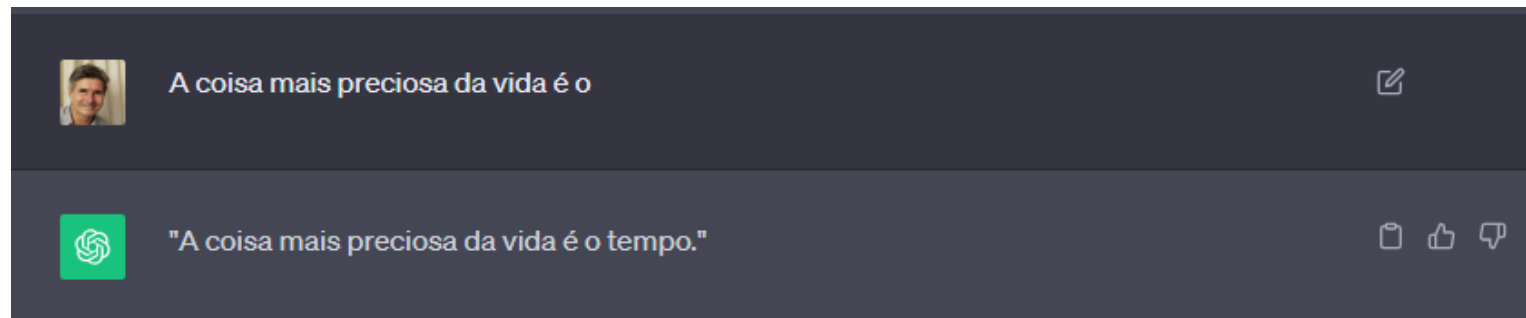
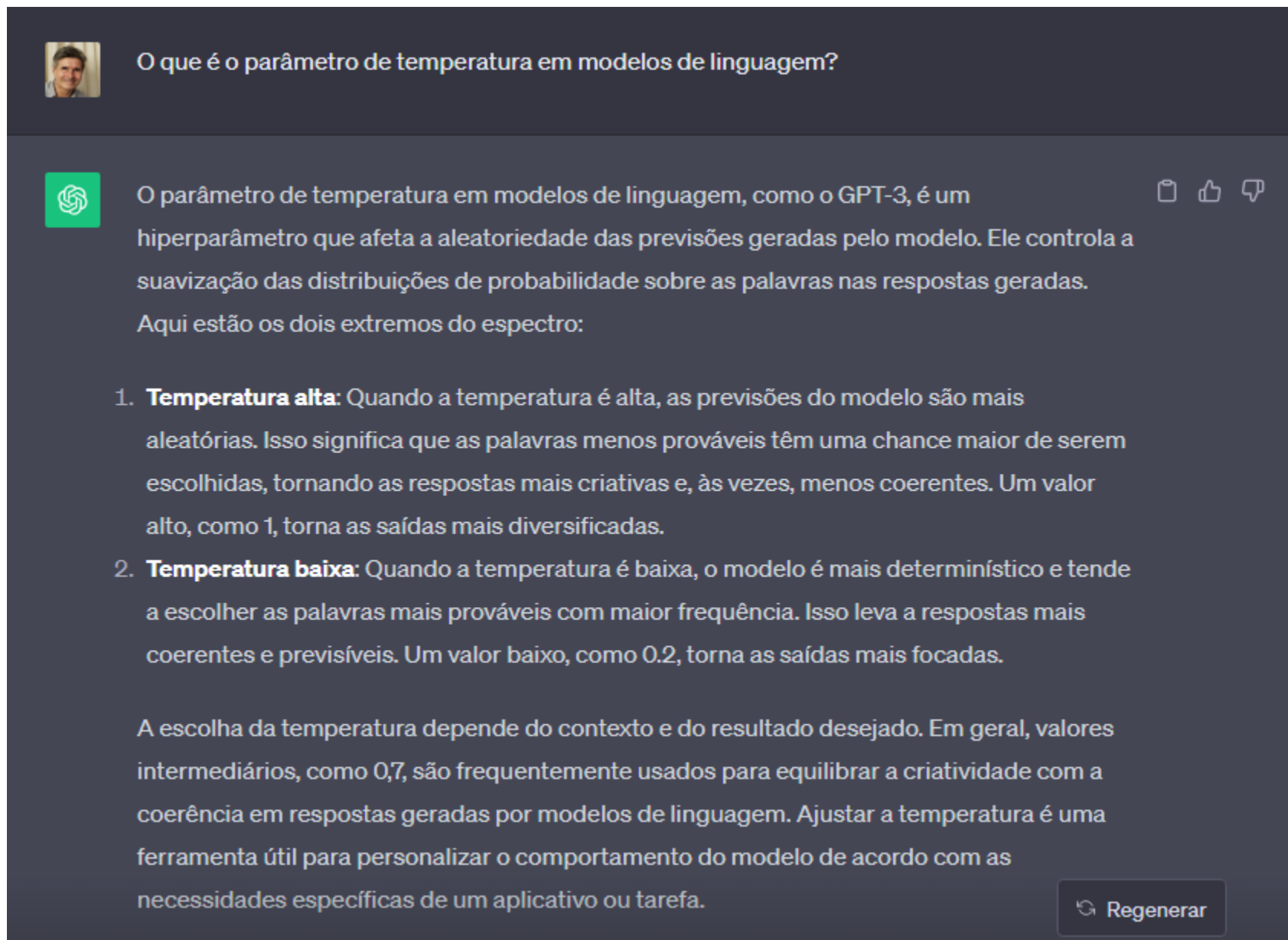


Figura 62

Para garantir maior **diversidade** nos conteúdos gerados, pode-se ajustar o parâmetro de "temperatura" [75, 78], um dos hiperparâmetros que controlam as saídas do modelo usado na função softmax que já mencionamos no BOX 3.

Como explicado pelo próprio ChatGPT, "valores mais altos de temperatura (como 0.7) produzem saídas mais aleatórias (mais criativas, mas talvez menos coerentes), enquanto "temperaturas menores" (como 0.2) fazem com que as saídas sejam as mais prováveis e frequentes (mais coerentes e previsíveis, e menos diversificadas).



O que é o parâmetro de temperatura em modelos de linguagem?

O parâmetro de temperatura em modelos de linguagem, como o GPT-3, é um hiperparâmetro que afeta a aleatoriedade das previsões geradas pelo modelo. Ele controla a suavização das distribuições de probabilidade sobre as palavras nas respostas geradas. Aqui estão os dois extremos do espectro:

1. **Temperatura alta:** Quando a temperatura é alta, as previsões do modelo são mais aleatórias. Isso significa que as palavras menos prováveis têm uma chance maior de serem escolhidas, tornando as respostas mais criativas e, às vezes, menos coerentes. Um valor alto, como 1, torna as saídas mais diversificadas.
2. **Temperatura baixa:** Quando a temperatura é baixa, o modelo é mais determinístico e tende a escolher as palavras mais prováveis com maior frequência. Isso leva a respostas mais coerentes e previsíveis. Um valor baixo, como 0.2, torna as saídas mais focadas.

A escolha da temperatura depende do contexto e do resultado desejado. Em geral, valores intermediários, como 0,7, são frequentemente usados para equilibrar a criatividade com a coerência em respostas geradas por modelos de linguagem. Ajustar a temperatura é uma ferramenta útil para personalizar o comportamento do modelo de acordo com as necessidades específicas de um aplicativo ou tarefa.

Regenerar

Figura 63

O hiperparâmetro "temperatura" não pode ser diretamente configurado usuário através de *Prompts*. Isso tem um lado bom, pois como vimos valores muito elevados de temperatura (como 0.9 por exemplo) introduzem maior aleatoriedade nas previsões, o que pode ter alguns riscos e efeitos indesejados. Dito de modo elegante, "com uma temperatura alta, o modelo é mais propenso a fazer escolhas menos prováveis, resultando em textos mais divergentes e menos coerentes, que podem divergem significativamente de conhecimentos prévios". Falando de forma mais simples, o modelo pode *alucinar* nas suas respostas. Pirar na batatinha. *Viajar na maionese*.

Embora não possa ser diretamente configurável pelo usuário através de Prompts, a "temperatura" pode ser ajustada programaticamente via API.

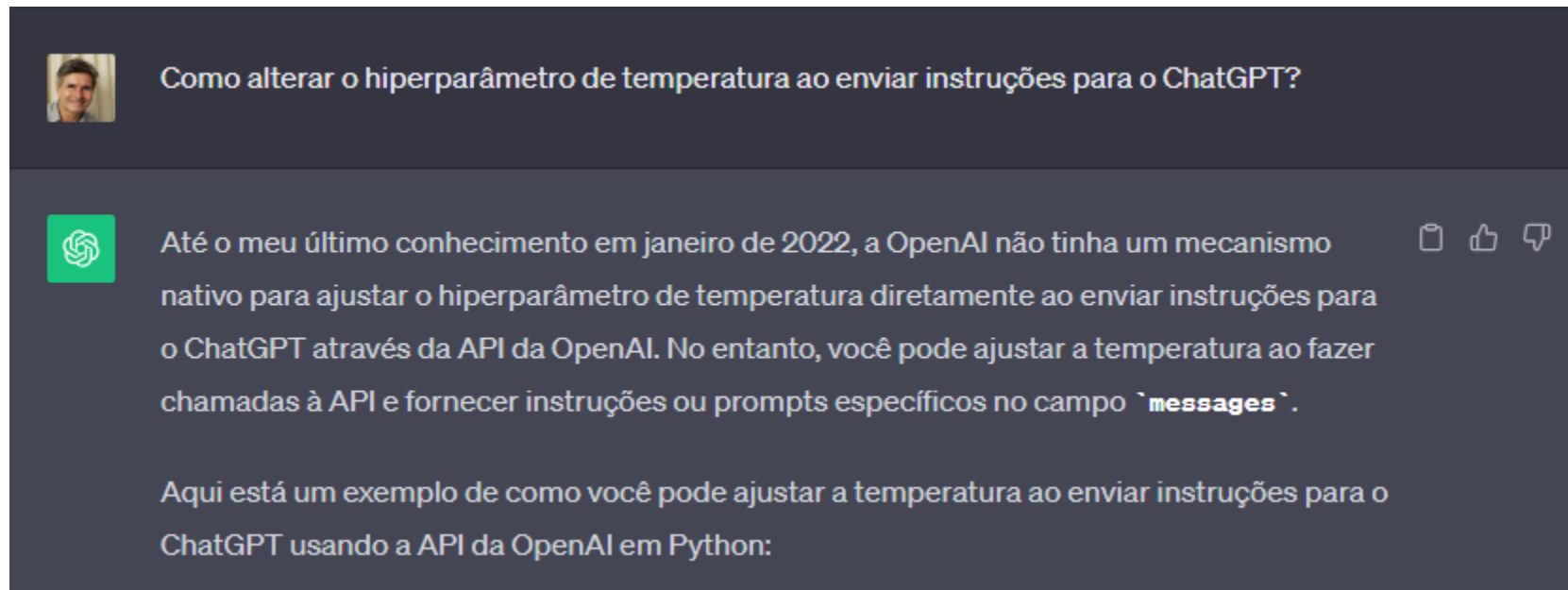
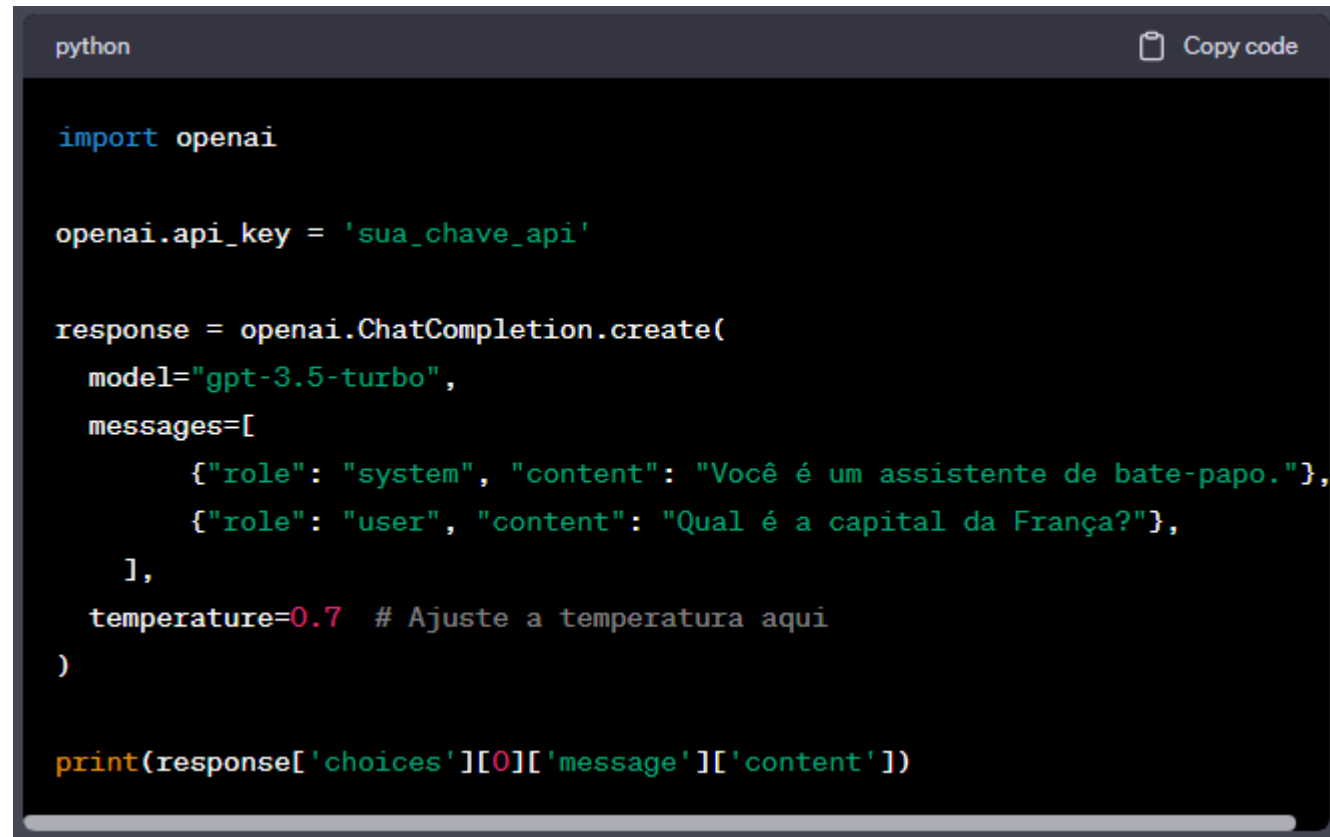


Figura 64

A Figura 65 mostra um código em Python para executar um *Prompt* com *ajuste de temperatura* via API (valor 0.7 no campo **temperature**). Este valor vai gerar respostas mais previsíveis do que um valor 0.9 por exemplo.



```
python                                                                    Copy code

import openai

openai.api_key = 'sua_chave_api'

response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "Você é um assistente de bate-papo."},
        {"role": "user", "content": "Qual é a capital da França?"},
    ],
    temperature=0.7 # Ajuste a temperatura aqui
)

print(response['choices'][0]['message']['content'])
```

Figura 65

No consumo de LLMs na modalidade MaaS (*Model-as-a-Service*), como na oferta da Microsoft de acesso nas APIs dos modelos da OpenAI, os custos dependem do modelo acessado e da quantidade de *tokens*, tanto nas instruções (*Prompts*) quando nas respostas geradas pelo modelo. Ver BOX 4.



### BOX 4. Quanto custa um token?

Nos modelos GPT da OpenAI a quantidade de *tokens* que pode ser gerada como resposta a uma requisição (*Prompt*) para o ChatGPT é limitada pelo parâmetro **max\_tokens**, e a cobrança pelo uso da API está vinculada ao número de *tokens*. O custo de um *token* depende do modelo que está em uso, sendo faturado em grupos de 1000 *tokens*.

Por exemplo, o preço cobrado pela Microsoft (Azure OpenAI) para acesso na API para uso do ChatGPT é de U\$ 0,003 para cada 1000 *tokens* no *Prompt*, e de U\$ 0,004 para cada 1.000 *tokens* na conclusão para o modelo GPT 3.5 Turbo, com 16k de contexto. Já para o modelo GPT-4 com 32k de contexto os custos são de U\$ 0,06 por 1.000 *tokens* no *Prompt*, e U\$ 0,12 por 1.000 *tokens* na conclusão. Estes preços são para acesso aos serviços da OpenAI hospedados no Azure na região Centro-Norte dos Estados Unidos, e podem variar para outras regiões.

Modelos	Contexto	Prompt (Por 1,000 tokens)	Conclusão (Por 1,000 tokens)
GPT-3.5-Turbo	4K	\$0,0015	\$0,002
GPT-3.5-Turbo	16K	\$0,003	\$0,004
GPT-4	8K	\$0,03	\$0,06
GPT-4	32K	\$0,06	\$0,12

Figura 66 - Fonte: [79]

Para mais informações consultar [79].

## 2.12. *ELMO, BERT e GPT*

Vimos na Seção 2.6 que um "*Word Embedding*" em um contexto do treinamento de modelos de linguagem é uma representação numérica de uma palavra em um espaço vetorial. Em termos simples, o *Embedding* é uma forma de representar dados em um formato que o modelo possa entender e manipular. Um problema no tipo de "*Embedding*" gerado pelo algoritmo **Word2Vec** [71] é que ele gera *Embeddings* fixos para palavras independentemente do contexto, ou seja, "banco" de tirar dinheiro e "banco" da praça teriam a mesma representação.

A partir de 2018 surgiram novos métodos capazes de gerar *Embeddings contextuais*, ou seja, representações vetoriais de palavras *que levam em consideração o contexto global da sequência*, resultando em representações mais ricas e adaptadas ao contexto. Estes *Embeddings contextuais* são vetores de números reais que representam palavras em um espaço vetorial de várias dimensões, e são utilizados nos modelos ELMo, BERT, GPT e vários outros.

### ***ELMo***

O ELMo (*Embeddings from Language Model*) [18] é um modelo de linguagem processado em RNNs (Redes Neurais Recorrentes) [64] do tipo LSTM (*Long Short-Term Memory*) [98]. Cada camada na rede pode conter uma representação vetorial diferente de cada *token* de entrada, gerada a partir de *Embeddings* contextuais. Além disso, em cada camada, o ELMo adota uma abordagem de contexto bidirecional, treinando representações diferentes para as informações à esquerda e à direita de cada palavra (as camadas da RNN "olham" para a frente e para trás na sentença). Assim, cada *token* pode ter várias representações ricas e contextualizadas, que são combinadas. O modelo trabalha com tokenização de palavras inteiras, e pode ser ajustado para tarefas específicas (por exemplo, prever a próxima palavra mais provável na sentença), aproveitando suas representações contextuais.

## **BERT**

Em 2018, o modelo BERT (*Bidirectional Encoder Representations from Transformers*) [19] foi introduzido por pesquisadores da Google.

O BERT utiliza uma estratégia de *máscara de atenção* para o pré-treinamento dos modelos de linguagem. Em vez da estratégia de "prever a próxima palavra", o BERT mascara aleatoriamente algumas palavras em uma sentença durante o treinamento e treina o modelo para prever essas palavras mascaradas. Isso permite que o modelo compreenda o contexto bidirecional da sequência.

O BERT melhora a atenção bidirecional introduzida pelo ELMo, pois introduz a atenção bidirecional *em uma única camada*, permitindo que cada palavra em uma sequência atue como uma palavra contextualizada para todas as outras, capturando contextos bidirecionais de maneira mais integrada.

Enquanto o ELMo utiliza tokenização de palavras completas, o BERT usa tokenização de subpalavras, muitas vezes implementada com o algoritmo BPE [66].

Uma vantagem do BERT em relação ao ELMo é a arquitetura da rede neural utilizada no treinamento. Enquanto o ELMo utiliza uma arquitetura LSTM bidirecional para capturar contextos, como mencionado, o BERT já usa a arquitetura *Transformer*, que é mais escalável e eficaz para modelar dependências de longo alcance (BOX 5).

Tanto o ELMo quanto o BERT são modelos inovadores, mas a abordagem bidirecional do BERT, juntamente com sua tokenização baseada em subpalavras, permitiu que ele capturasse relações mais complexas e dependências de longo alcance entre palavras, em comparação com modelos anteriores como o ELMo.

## GPT

Enquanto o BERT utiliza treinamento bidirecional em dados mascarados para entender o contexto de uma palavra em relação a todo o contexto da sentença, os modelos GPT (*Generative Pre-trained Transformer*) da Open AI são treinados para prever a próxima palavra em uma sequência unidirecional.

Um modelo GPT (como o GPT-3) é treinado em grandes conjuntos de dados não rotulados antes de ser afinado com supervisão humana, e emprega múltiplas camadas de decodificadores *Transformer*, cada uma composta por mecanismos de Atenção e redes neurais totalmente conectadas. O resultado é um modelo altamente capaz na geração de texto coerente e na realização de diferentes tipos de tarefas de processamento de linguagem natural, sobretudo a tradução de idiomas e a geração de textos sintéticos.

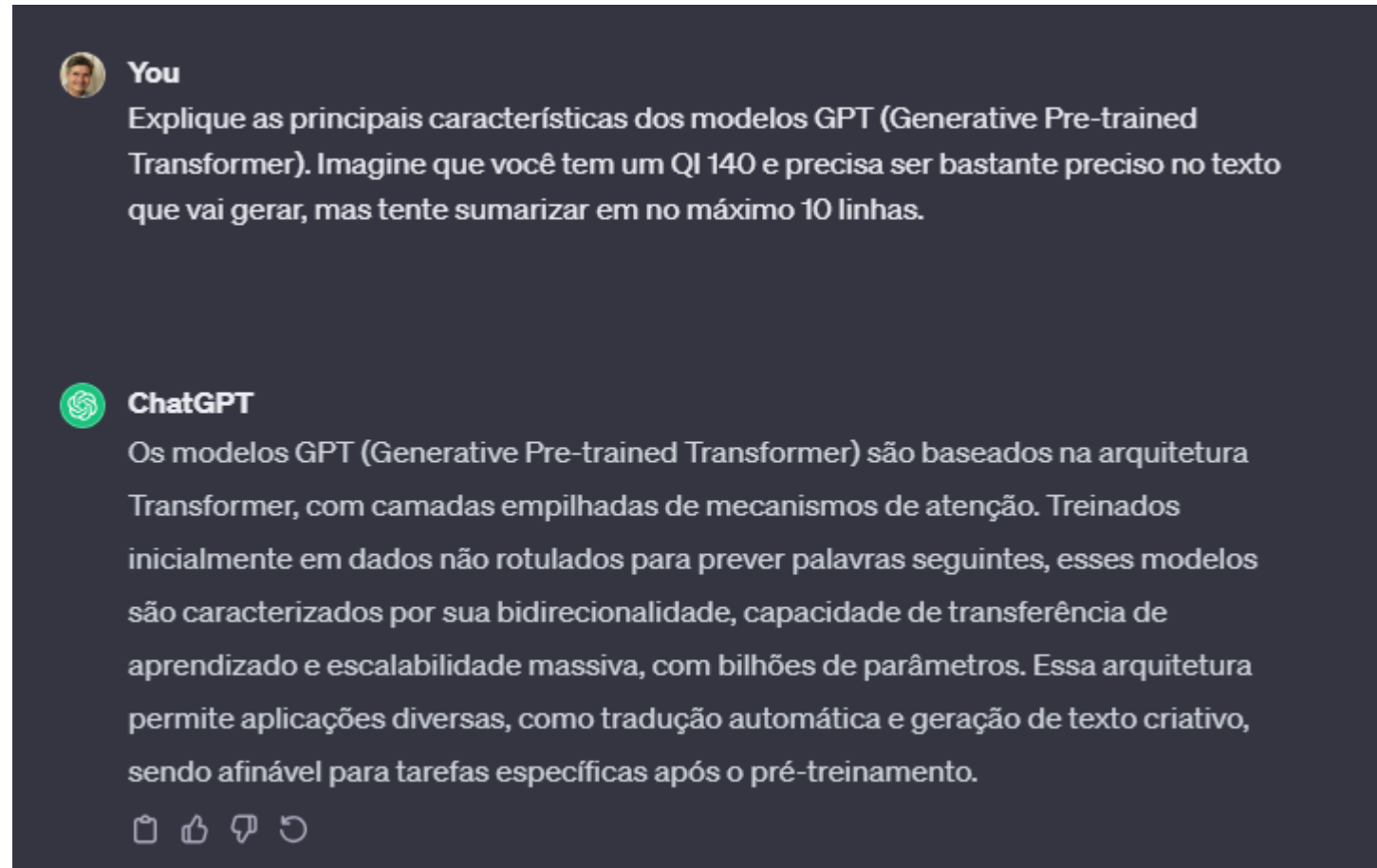


Figura 67

## BOX 5. Transformers e *Embeddings* Posicionais

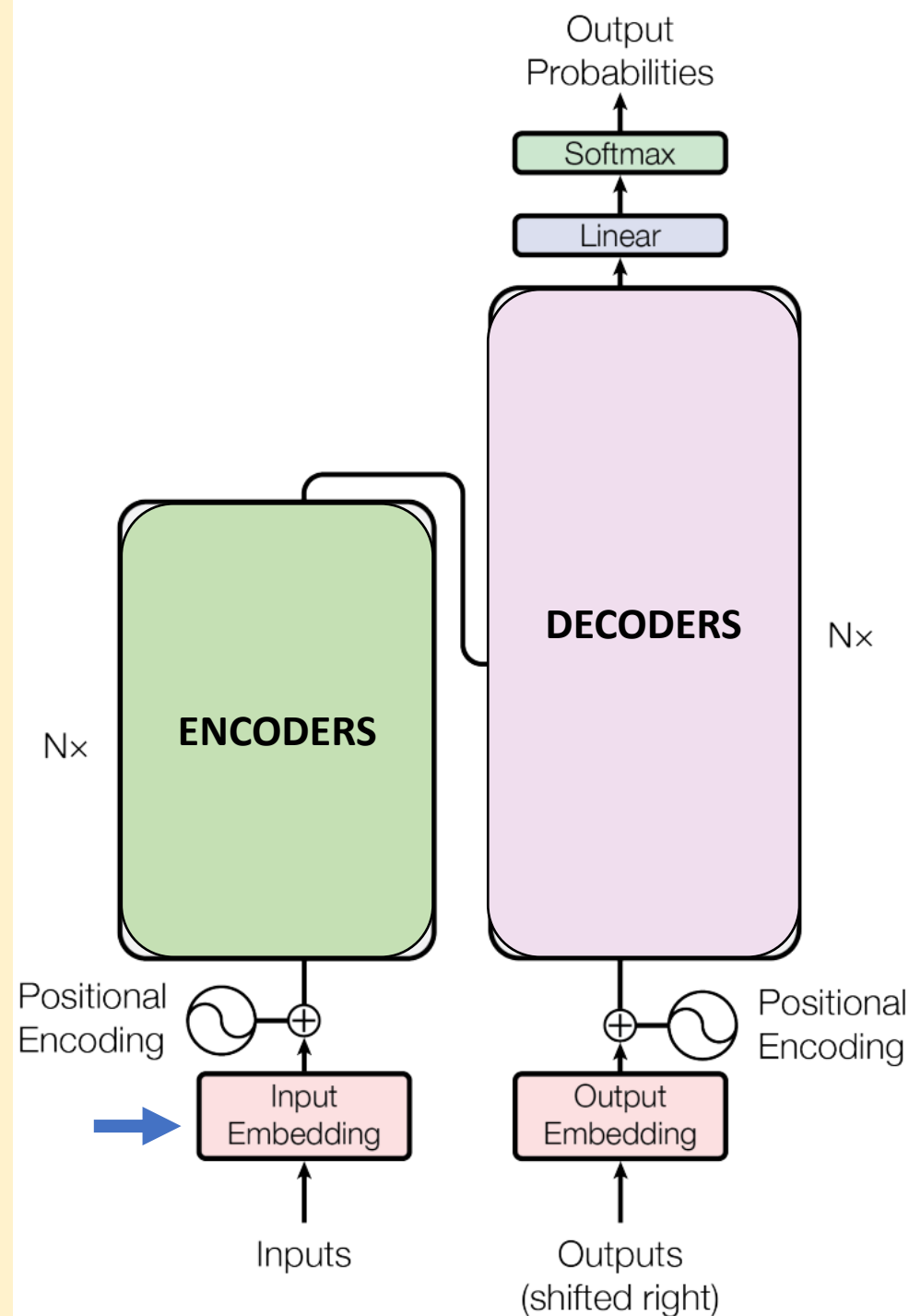
A Figura 68 adaptada de [80] mostra uma versão simplificada da arquitetura *Transformer*, redes neurais utilizadas no treinamento de modelos de linguagem.

Embora a arquitetura *Transformer* seja conhecida por seu mecanismo de Atenção e sua capacidade de processar sequências de entrada sem depender de estruturas sequenciais fixas, ela ainda faz uso de *Word Embeddings*, mas não são mais os mesmos "*Embeddings*" gerados pelo *Word2Vec*.

Em vez disso, são utilizados "*Embeddings Posicionais*".

Por hora, os detalhes do "*Embedding Posicional*" e do que há dentro dos blocos com ENCODERS e DECODERS do *Transformer* (várias redes neurais em cada um) são omitidos em favor da clareza. A arquitetura será discutida no Capítulo 4, mas precisamos nos preparar para esta discussão analisando primeiro como é feito o treinamento dos grandes modelos de linguagem.

Figura 68 - Fonte: [80]



# Capítulo 3 - Treinando Assistentes GPT

Com o que aprendemos nos Capítulos 1 e 2 já estamos bem equipados para entender como um grande modelo de linguagem é treinado.

O treinamento dos modelos GPT da OpenAI evoluiu muito desde o GPT-1 até o lançamento do modelo GPT-3 em 2020. Inicialmente, o processo envolvia apenas duas etapas: um pré-treinamento não supervisionado em um extenso conjunto de dados e um ajuste fino supervisionado para otimização em tarefas específicas de NLP.

Com o aprendizado adquirido no treinamento destas versões, sobretudo após o GPT-2, os cientistas da OpenAI descobriram que era possível melhorar o treinamento dos modelos através de instruções, chamadas de *Prompts*, fornecidas por especialistas humanos.

Após o lançamento do modelo GPT-3, vieram os modelos da família GPT-3.5. Além das etapas de pré-treinamento e aprendizado por reforço inicial, foram introduzidas duas novas etapas no treinamento. Na etapa 3 é gerado um modelo de recompensas (RM), e na etapa 4 é produzido o modelo final. Nestas duas etapas finais do *Pipeline* de treinamento, o uso de *Prompts* com supervisão humana ou RLHF (*Reinforcement Learning from Human Feedback*) desempenha um papel crucial, melhorando o "alinhamento" entre as respostas do modelo e as expectativas dos usuários. O modelo resultante deste novo processo de treinamento foi chamado InstructGPT, e superou significativamente o modelo GPT-3 de 175 bilhões de parâmetros treinado sem RLHF.

O modelo GPT-3.5 treinado com RLHF é utilizado na versão gratuita do Assistente ChatGPT. A versão paga utiliza o modelo GPT 4.0, que também é RLHF (veja a Figura 3).

A fonte primária de consulta para este capítulo é a excelente palestra de Andrej Karpathy no evento Microsoft Build 2023 (State of GPT | BRK216HFS). Os que tiverem interesse em assistir o video original podem consultar [29] em Referências. Karpathy já foi director de IA na Tesla, é um dos membros fundadores da OpenAI e é um educador na Universidade de Stanford.



Figura 69 - Fonte: [29]



### 3.1. Pipeline de treinamento de Assistentes GPT

Como explicado em [29], o treinamento do modelo utilizado por um *Assistente GPT* como o ChatGPT (como o GPT 3.5) é um processo sequencial composto por 4 estágios (Figura 70). Em cada estágio, redes neurais com arquitetura *Transformer* são utilizadas para treinar novos modelos com diferentes conjuntos de dados (*Datasets*). No primeiro estágio, o treinamento é não supervisionado e o modelo resultante é o **Modelo Base (MB)**. Nos três estágios seguintes o treinamento é supervisionado por humanos, visando a redução de erros e a geração de modelos cada vez mais precisos, isto é, com melhor performance (**Modelo SFT** -> **Modelo RM** -> **Modelo RL**). Os estágios 3 e 4 combinados formam o RLHF (*Reinforcement Learning from Human Feedback*) [83]. O *Pipeline* completo é mostrado na Figura 71.

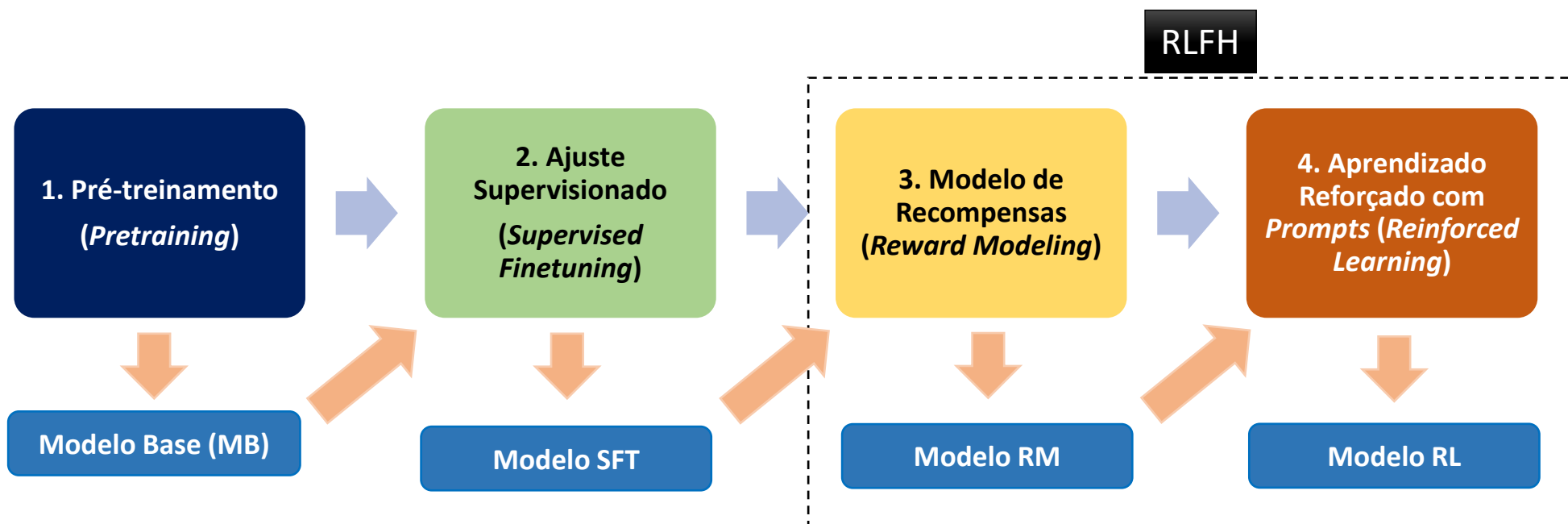


Figura 70 - Fonte: Adaptado de [29] pelo autor



Pipeline de treinamento de Assistentes GPT

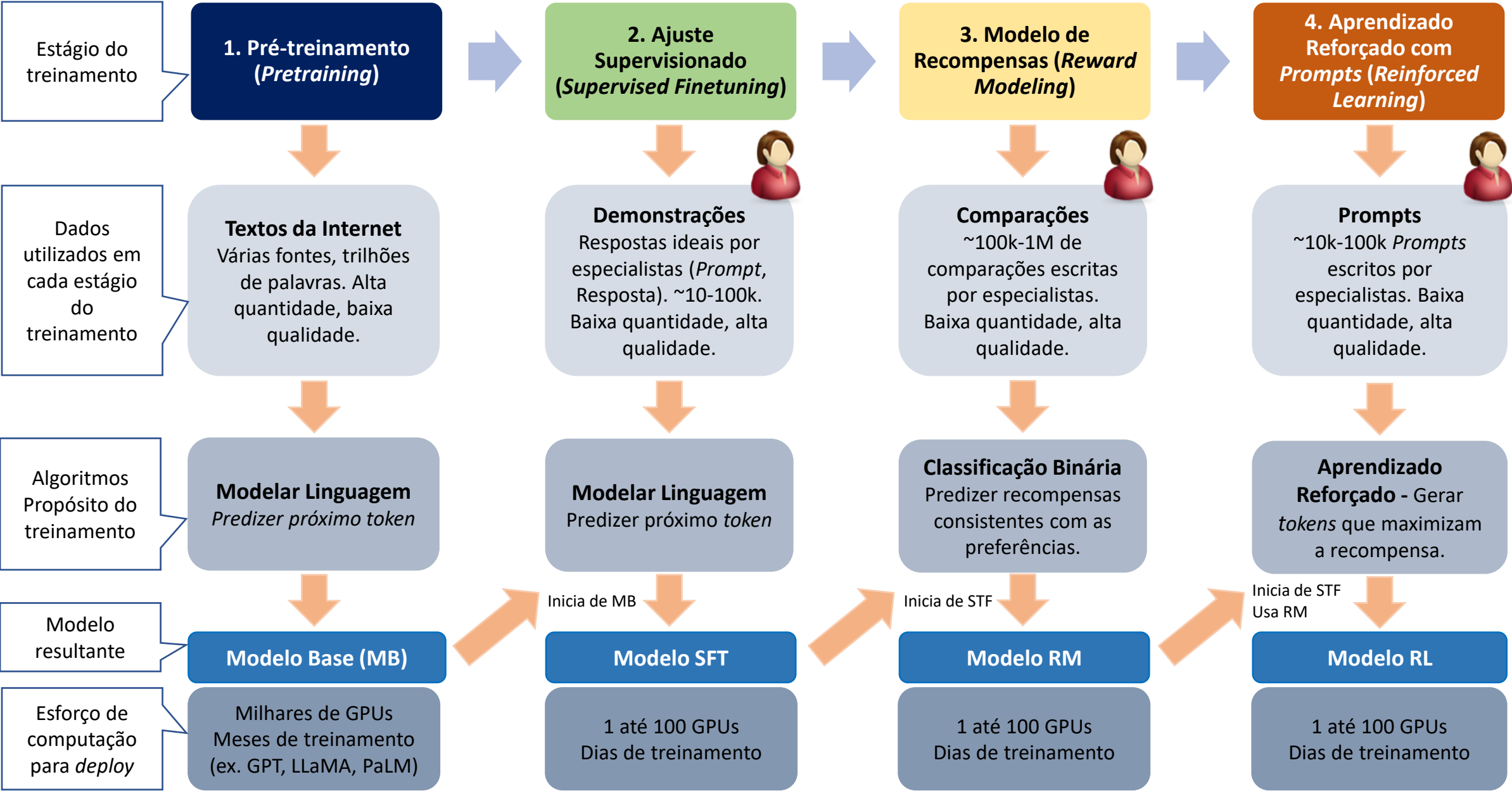
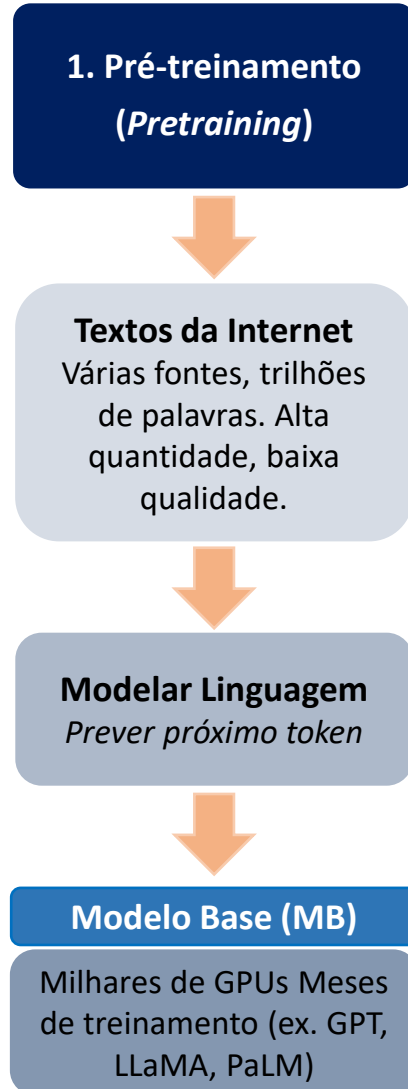


Figura 71 - Fonte: Adaptado de [29] pelo autor

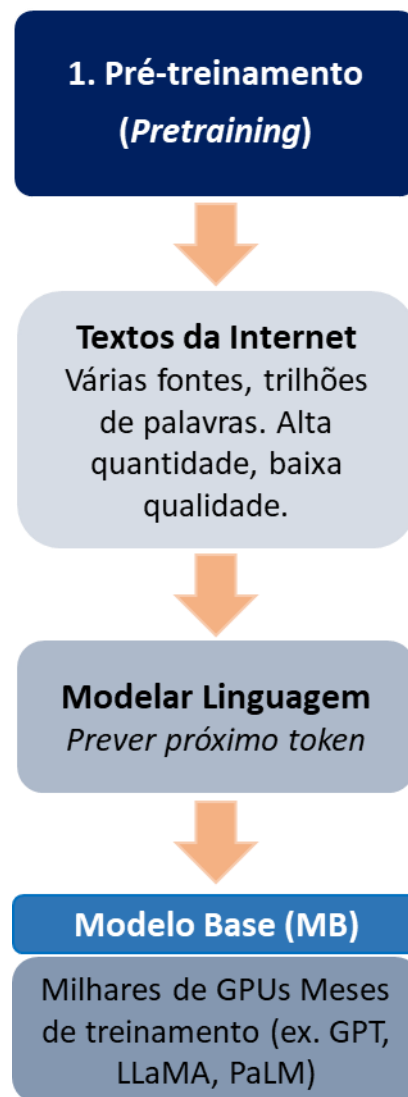
## Estágio 1 - Pré-treinamento



No Pré-treinamento são utilizadas quantidades massivas de textos de linguagem natural (TRILHÕES de palavras, extraídas de *datasets* públicos da Internet) para treinar um modelo de linguagem de modo não supervisionado. Durante o treinamento por redes neurais *Transformer* o modelo vai aprendendo sobre os padrões e estruturas da linguagem, até se tornar capaz de gerar novo conteúdo (texto), desde pequenas frases e parágrafos até artigos científicos ou livros inteiros. Embora a quantidade de dados utilizados no treinamento seja imensa neste estágio, a qualidade é baixa.

Nesta etapa ocorre a **tokenização**, onde as palavras da linguagem natural são convertidas em *tokens*. Depois, ocorre a **vetorização**, o processo de *Embedding* Posicional, onde os *tokens* são representados por vetores de números reais somados a outros vetores que controlam a posição de cada *token* na sequência. Como vimos, palavras com significado próximo na linguagem natural (como "mulher" e "mãe") tendem a gerar representações vetoriais próximas. Desta forma, o modelo vai aprendendo em cada iteração não apenas a estrutura gramatical da linguagem, mas também os relacionamentos implícitos entre palavras, e assim se torne cada vez melhor em "prever o próximo *token*", ou sugerir a próxima palavra para completar uma sentença.

O modelo gerado ao final do Estágio 1 é o **Modelo Base (MB)**. *Cerca de 99% do esforço de computação ocorre neste estágio*, que requer supercomputadores com milhares de GPUs, e ainda assim são necessários meses de treinamento, ao custo de milhões de dólares.



No Pré-treinamento de modelos de linguagem são utilizados diversos *Datasets* obtidos na Internet. A Tabela mostra algumas destas fontes. Por exemplo, o *Common Crawl* é um *Dataset* imenso, de onde podem ser extraídos dezenas de bilhões de *tokens*.

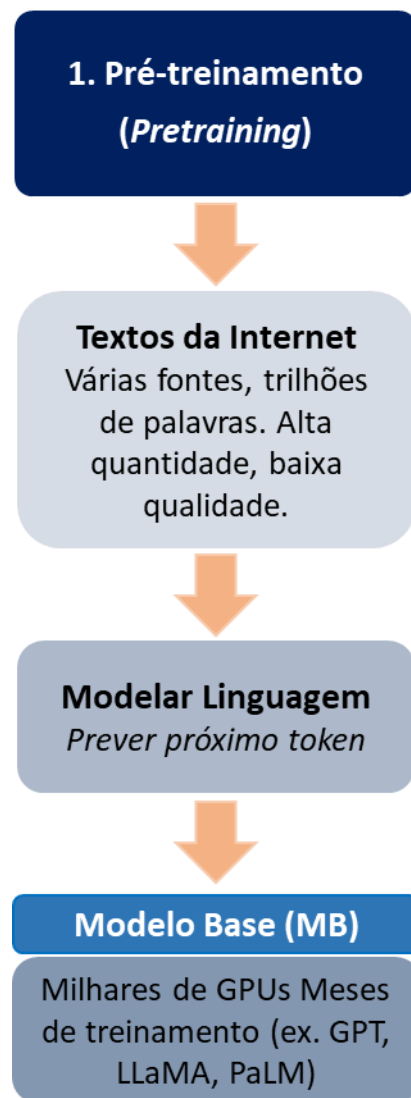
Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Figura 72 - Fonte: [29]

No caso dos modelos GPT, para melhor desempenho os cientistas da OpenAI filtraram os dados do *Common Crawl* antes de utilizá-los, visando melhorar a qualidade dos dados, por isso há a indicação *Common Crawl (filtered)* na primeira linha.

Os dados de todas estas fontes são misturados e é feito um *sampling* para extrair dados de cada *Dataset* (conforme certa proporção) para formar um *Dataset* único, que será efetivamente utilizado no treinamento do Modelo Base (MB).

No exemplo, o *Dataset* resultante da filtragem do *Common Crawl* tem peso 60% no mix de dados utilizados para o treinamento, enquanto a Wikipedia tem peso 3%.



Antes de iniciar o treinamento do modelo com o *Dataset* unificado é preciso executar um mais passo de pré-processamento - a **tokenização**. Um algoritmo como o BPE (*Byte Pair Encoding*) [66] é utilizado para converter todas as palavras do *Dataset* de Pré-treinamento em *tokens*, gerando um **vocabulário** ou **dicionário de tokens**.

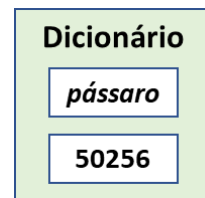


Figura 73 - Cada token no dicionário é identificado por um número inteiro. São estes números que são passados para a rede neural Transformer.

Figura 73 - Fonte: Adaptado de [29] pelo autor

A Figura 74 mostra a escala dos hiperparâmetros que ocorrem no estágio de Pré-treinamento de grandes modelos de linguagem. Como os dados para o GPT-4 ainda não foram liberados pela OpenAI, são usados os dados para o GPT-3, um modelo de 2020 (de três anos atrás!), e também para o modelo LLaMA da MetaAI, que é bem mais recente (2023).

2 example models			
GPT-3 (2020)	50,257 vocabulary size	LLaMA (2023)	32,000 vocabulary size
	2048 context length		2048 context length
	175B parameters		65B parameters
	Trained on 300B tokens		Trained on 1-1.4T tokens

Figura 74 - Fonte: [29]

### GPT-3

- O **vocabulário** para o GPT-3 tinha **50.257 tokens diferentes**.
- O **contexto** tinha **2.048 tokens**. O **contexto** é um hiperparâmetro bastante importante, que define a quantidade de *tokens* que o modelo vai levar em conta para prever o próximo *token* em uma sequência. Já há modelos que utilizam até 100.000 *tokens* como contexto no pré-treinamento.
- O **número de parâmetros** do modelo GPT-3 é **175 B (bilhões)**. O número de parâmetros tem relação com a quantidade de pesos utilizados nas conexões entre neurônios nas diversas camadas das redes neurais. Estes pesos são ajustados durante o treinamento.
- O modelo foi treinado com **300 bilhões de tokens**.

### LLaMa

- **Vocabulário de 32.000 tokens**.
- **Contexto com 2.048 tokens**.
- **Número de parâmetros = 65 B (bilhões)**.
- Modelo treinado mais de **um trilhão de tokens** (1T - 1.4T), durante 21 dias com cerca de 2.000 GPUs.

### *Tamanho de modelo não é documento*

Como explicado por Andrej Karpathy em [29], embora o LLaMA seja um modelo "menor" que o GPT-3 (65 bilhões de parâmetros *versus* 175 bilhões), trata-se de um modelo bem mais avançado e com performance melhor. Isto se deve ao fato de que foi treinado com uma quantidade de *tokens* muito maior (acima de 1 trilhão).

Muito bem, temos então *300 bilhões de tokens* gerados a partir de documentos da Internet para treinar o GPT-3 para treinar o modelo.

Como estes dados são passados como INPUT para o ENCODER da rede *Transformer* no Pré-treinamento?

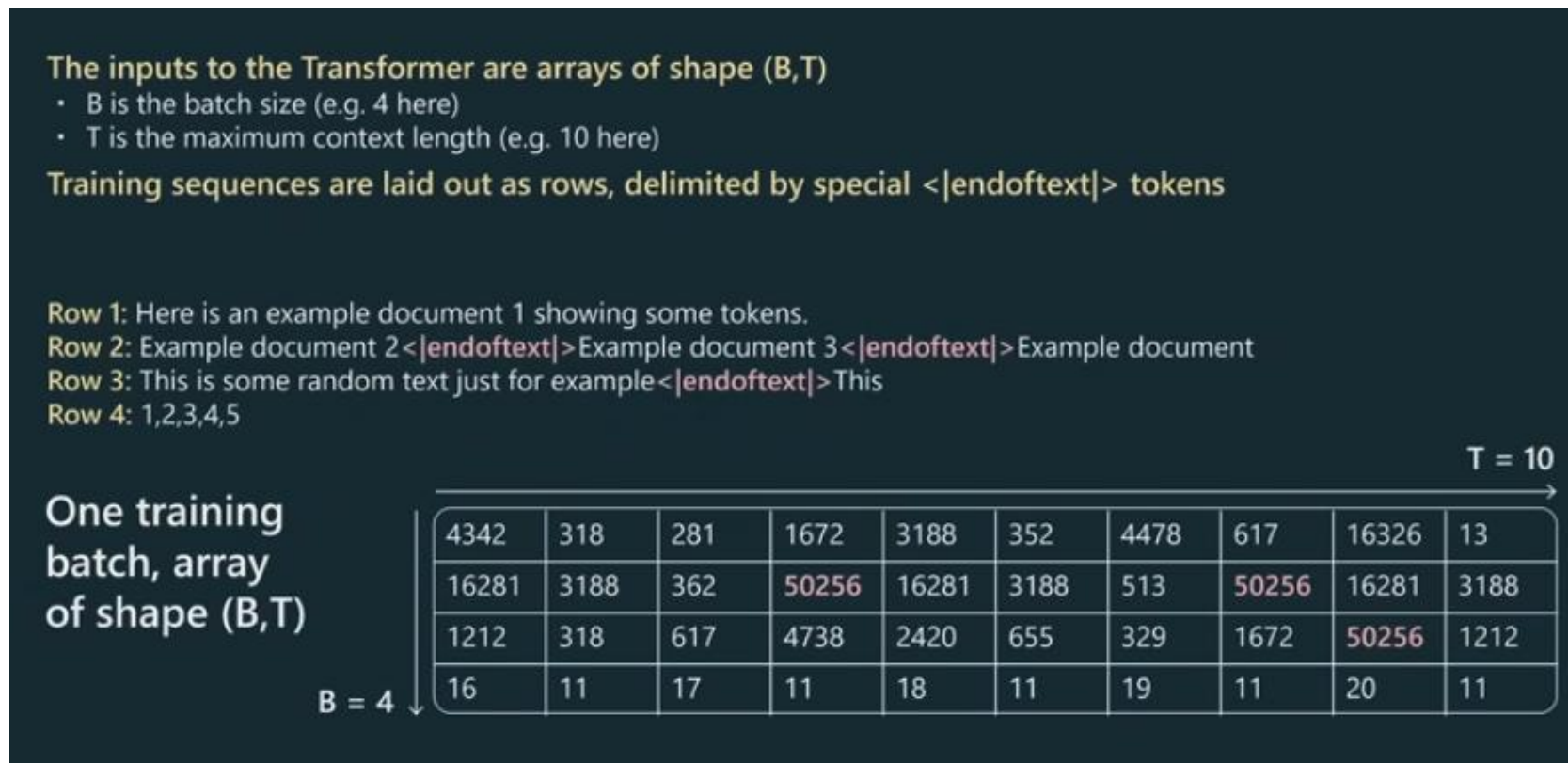


Figura 75 - Fonte: [29]

Figura 75 - Os INPUTS para o Transformer são matrizes de tokens (mais precisamente, matrizes com os identificadores numéricos dos tokens), contendo B linhas e T colunas, onde B é o tamanho do Batch (B = 4 no caso do exemplo) e T é o tamanho do contexto (no GPT-3, T = 2.048 tokens). Para que a matriz possa ser visualizada, foi utilizado T=10 como simplificação. A matriz contém os tokens dos vários documentos usados no treinamento, separados por um token especial `[endoftext]`, no caso, **50256**. Assim, por exemplo, o documento 1 vai do token 4342 até o token 362. Esta matriz (B, T) é um "Batch de treinamento".



Os "Batches de treinamento" contendo os identificadores numéricos dos *tokens* que formam os diversos documentos usados no treinamento são passados como INPUT para o bloco de ENCODER da rede *Transformer*, onde são VETORIZADOS, e é iniciado o processo iterativo de treinamento, através de mecanismos de Atenção.

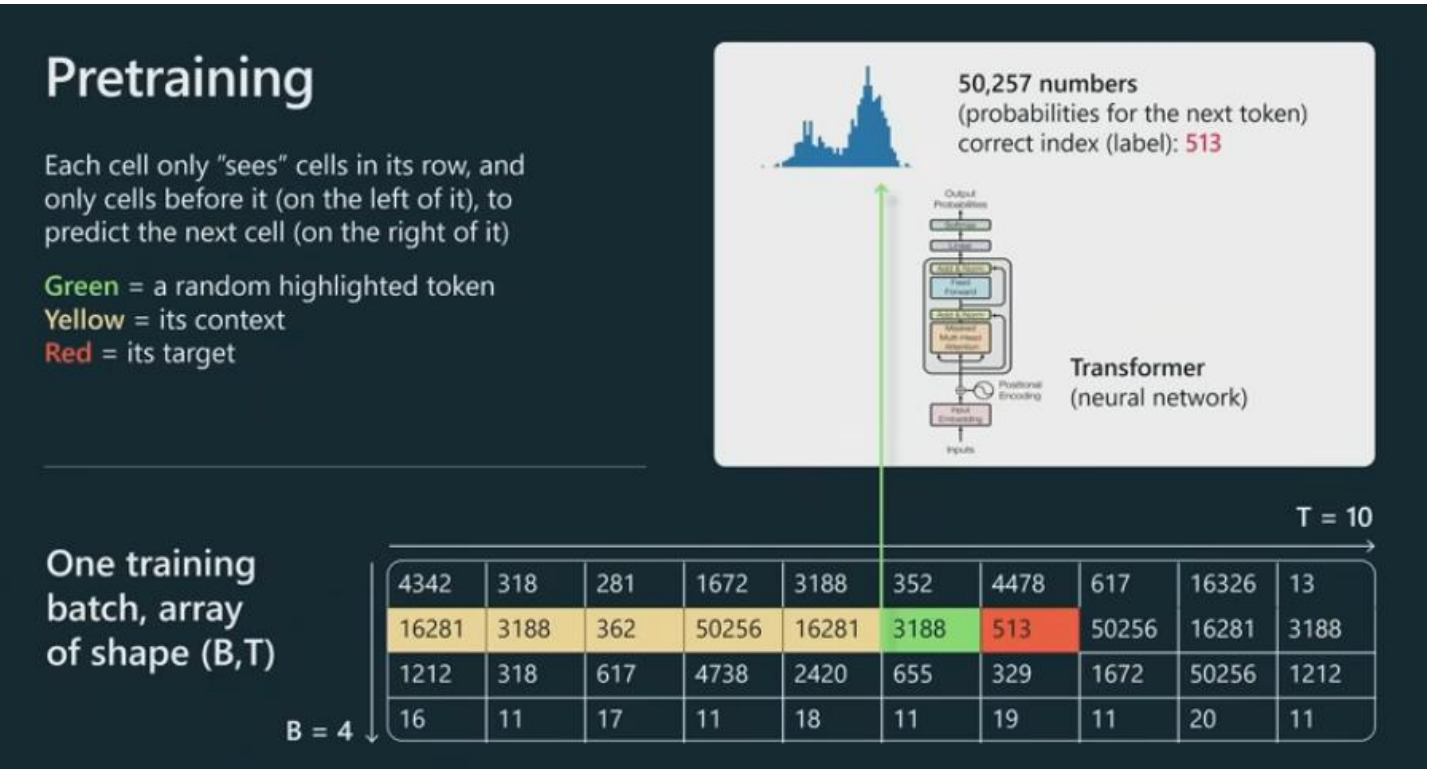


Figura 76 - Fonte: [29]

Na Figura 76 a linha em destaque tem um *token* em verde (ID 3188). Este *token*, bem como todos os *tokens* em amarelo que vieram antes (*tokens* de contexto), são passados como INPUT para a rede *Transformer*. A célula em vermelho é o "alvo", ou o *token* previsto como o mais provável para continuar a sequência. A rede neural calcula **probabilidades** para cada um dos 50.257 *tokens* do vocabulário. No exemplo, o modelo prediz que o *token* ID 513 é o **mais provável para completar a sequência**. Este processo se repete em paralelo em todas as células de diferentes *Batches* de treinamento - *predizer o próximo token*, com base no contexto anterior.

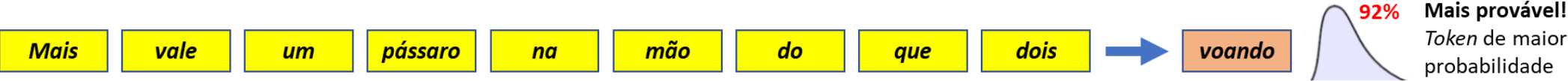



Figura 77

A tabela extraída de [15] mostra parâmetros de configuração da rede *Transformer* no treinamento de diferentes versões do modelo GPT-3. A última linha se refere ao modelo GPT-3.



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	$0.6 \times 10^{-4}$

**Table 2.1:** Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

Figura 78 - Fonte: [15]

- O modelo foi treinado com 175 bilhões de parâmetros ( $n_{\text{params}}$ ).
- A rede neural utilizada no treinamento tinha 96 camadas ( $n_{\text{layers}}$ ).
- Os parâmetros  $n_{\text{head}}$  e  $d_{\text{head}}$  estão relacionados com as **cabeças de Atenção** da rede.
- O tamanho de cada *Batch* de treinamento (*Batch Size*) também é indicado (3.2 milhões de *tokens*).

No Capítulo 4 vamos estudar a arquitetura *Transformer* com maior detalhamento, abrindo as "caixas pretas" dos blocos de ENCODER e DECODER. Se você chegou até aqui, prossiga na leitura pois o assunto é altamente interessante.



# BOX 6. Aprendendo a completar um texto de Shakespeare

Experimento feito com o ChatGPT pelo *The New York Times* [81]. Neste experimento foi utilizado um modelo *Small GPT* (com apenas 125 milhões de parâmetros, segundo a tabela da página anterior) - um modelo bem pequeno capaz de aprender o básico da linguagem

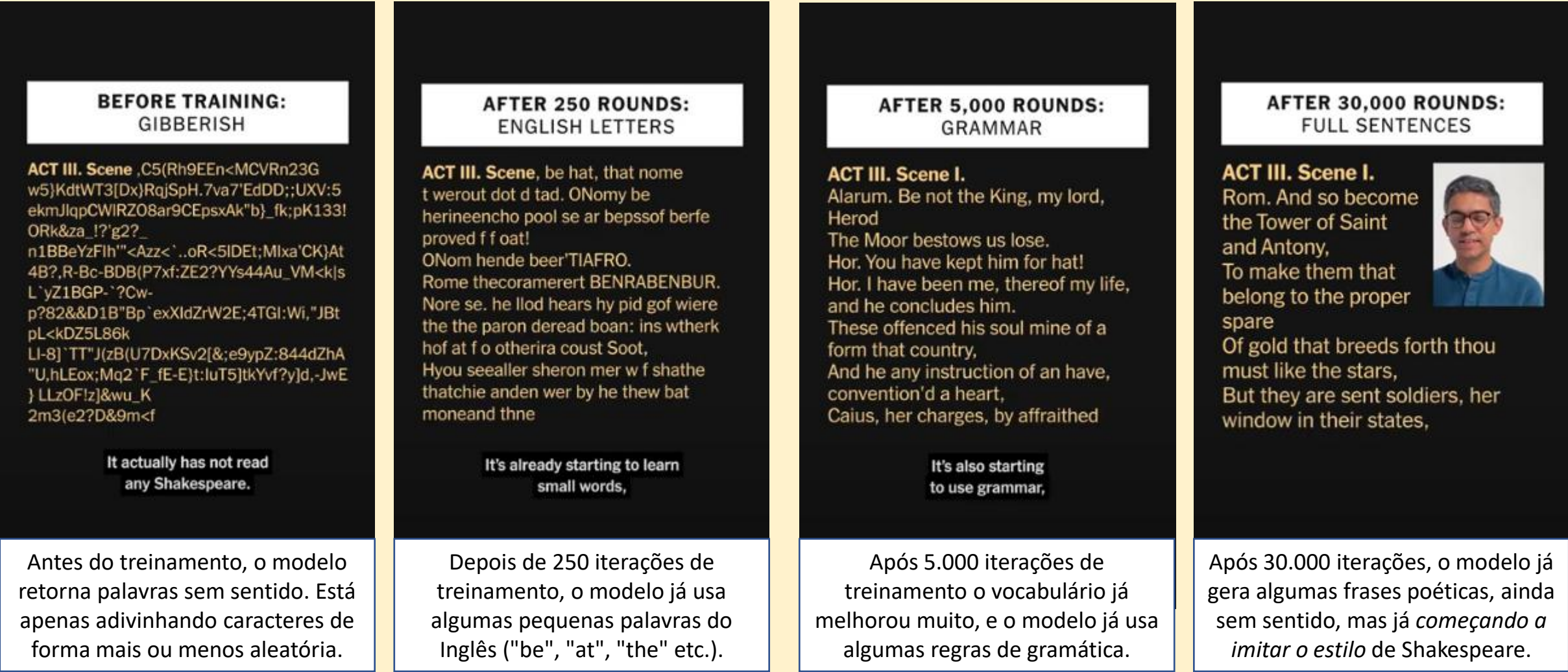


Figura 79 - Fonte: [81]

Como vimos, ao final do Estágio 1 (que é não supervisionado) é gerado um Modelo Base (MB) de linguagem. Porém, **MODELOS BASE DE LINGUAGEM AINDA NÃO SÃO ASSISTENTES**. Embora mesmo estes Modelos Base que resultam do Estágio 1 possam ser "enganados" para responder perguntas ou executar outras tarefas através da *Engenharia de Prompts*, para criar um verdadeiro Assistente como o ChatGPT é preciso muito mais treinamento, e com supervisão humana - que nos leva ao próximo Estágio do *Pipeline*, que é o *Supervised Finetuning*.

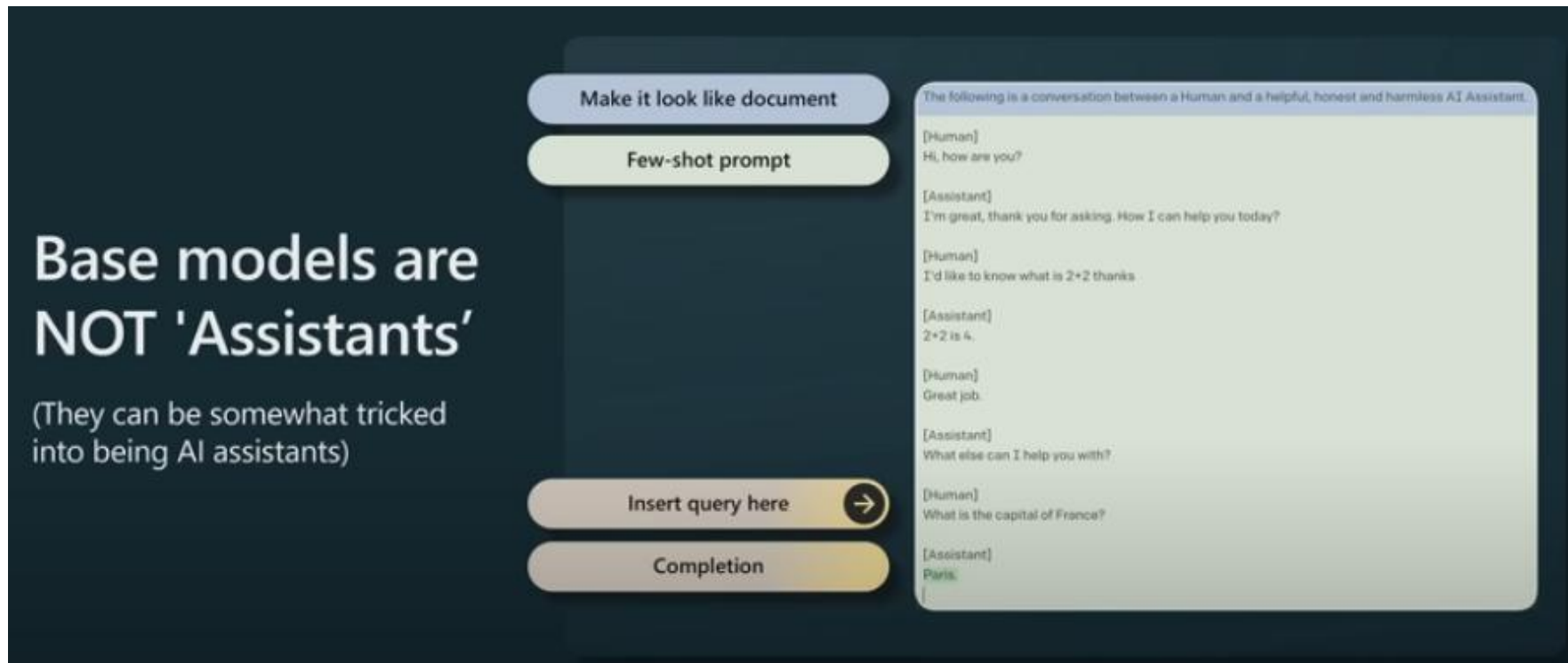


Figura 80 - Fonte: [29]

## Estágio 2 - Ajuste Supervisionado via Prompts

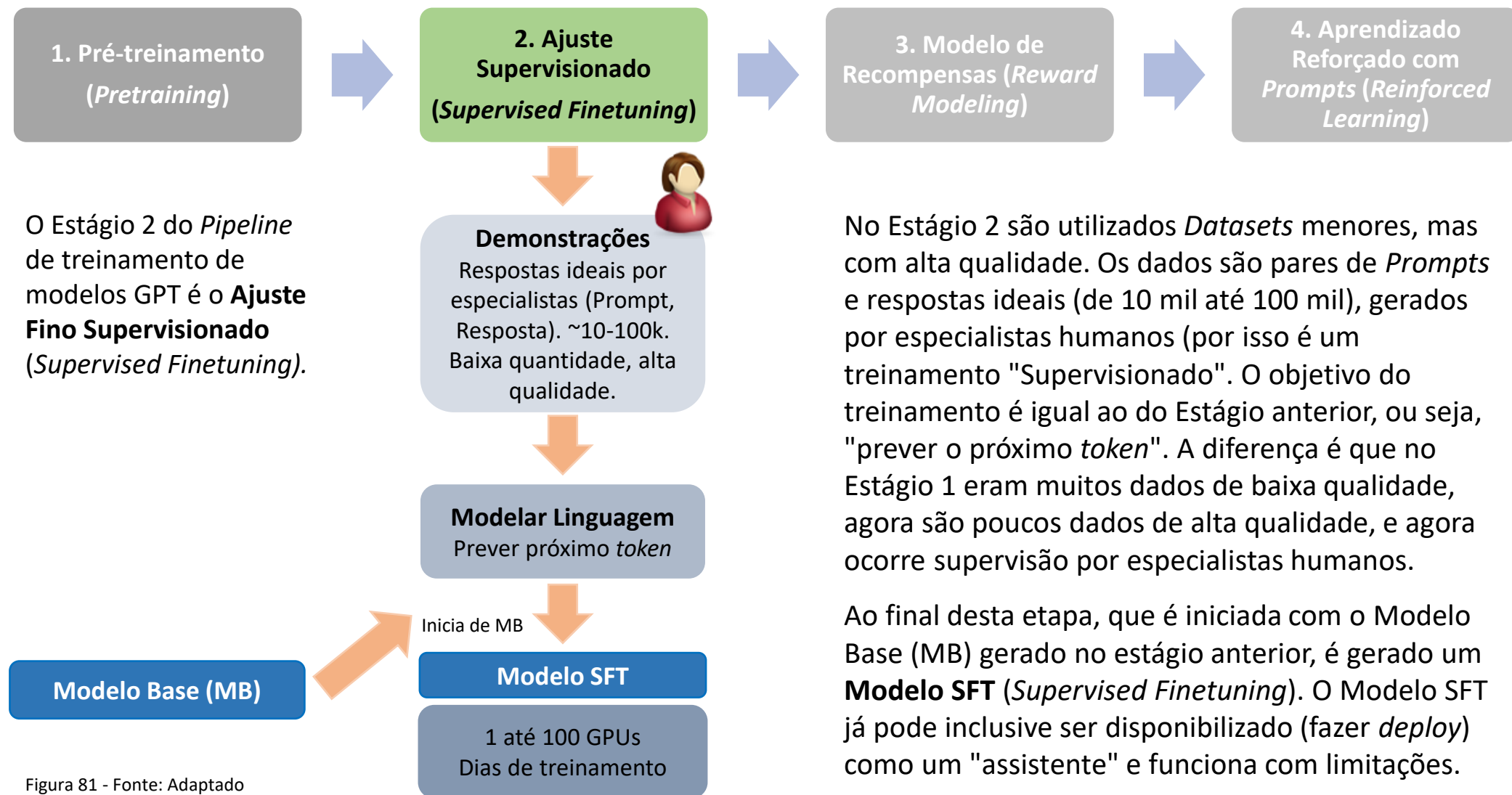


Figura 81 - Fonte: Adaptado de [29] pelo autor

No Estágio 2 de Ajuste Supervisionado, o **Modelo SFT** é treinado com dados que passam pela curadoria de humanos. Estes dados contêm dezenas de milhares de *Prompts* e as suas "respostas ideais" (as respostas que gostaríamos que o LLM gerasse). O modelo "aprende" a replicar o estilo dos exemplos durante o treinamento.

A Figura 82 mostra um exemplo de *Prompt* - Resposta utilizado no treinamento do Modelo SFT do LLaMA 2-Chat [82], que foi *otimizado para conversar com seres humanos*. As rimas se perdem na tradução mas funcionam perfeitamente em Inglês.

*Resposta:*  
Hidrogênio vem primeiro, como elemento número um.  
Hélio é o segundo para que os balões se divirtam!  
Em terceiro vem o Lítio, para carregar as baterias,  
Seguido pelo Berílio em esmeraldas pequenas ou grandes.  
O Boro é o número cinco, para nos ajudar a limpar as coisas,  
O Carbono vem em sexto, e aparece em todos os genes.  
O Nitrogênio é o sétimo, em cada respiração que fazemos,  
Produzindo mais do que o oito (Oxigênio) na atmosfera.  
O número nove é o Flúor, que nos ajuda a limpar os dentes.  
O Neon nos dá sinais luminosos que podemos visualizar.

3.1 Supervised Fine-Tuning (SFT)

*Prompt: Escreva um poema para me lembrar os 10 primeiros elementos da tabela periódica, com cada elemento em sua própria linha.*

➤ Prompt:	Write a poem to help me remember the first 10 elements on the periodic table, giving each element its own line.
Response:	Hydrogen comes first as element number one. Helium is second for balloons to have fun! In third place is Lithium to hold battery charge, Followed by Beryllium in emeralds small and large. Boron's number five to help us keep things clean. Carbon's next at six, and it's found in every gene. Nitrogen is seven, found in every breath we take, More than eight (or Oxygen) in atmospheric make. Number nine is Fluorine, helping polish up our teeth. Neon gives us glowing signs to read and stand beneath.

Figura 82 - Fonte: [82]

### Estágio 3 - Modelo de Recompensas (RM)

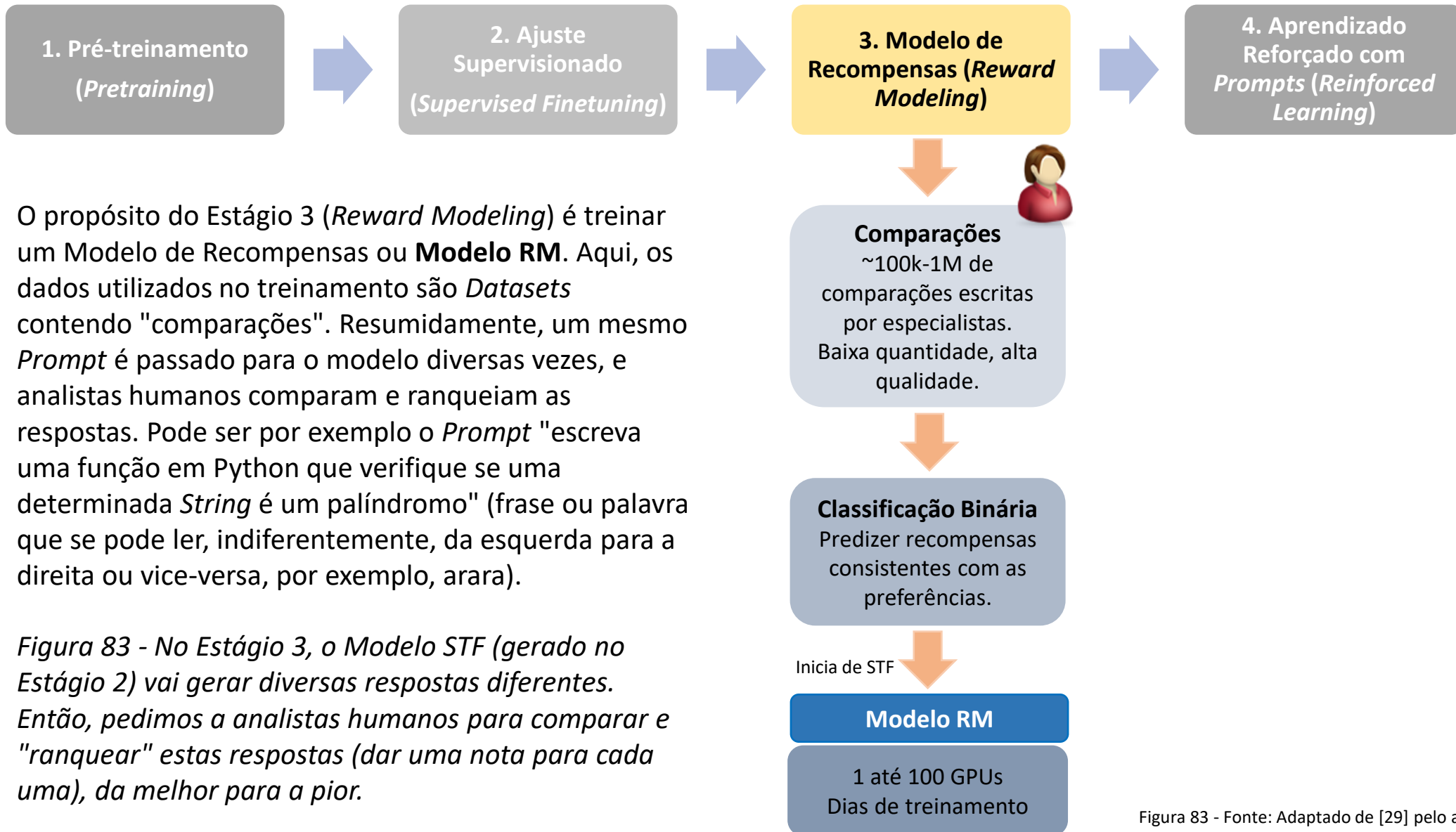
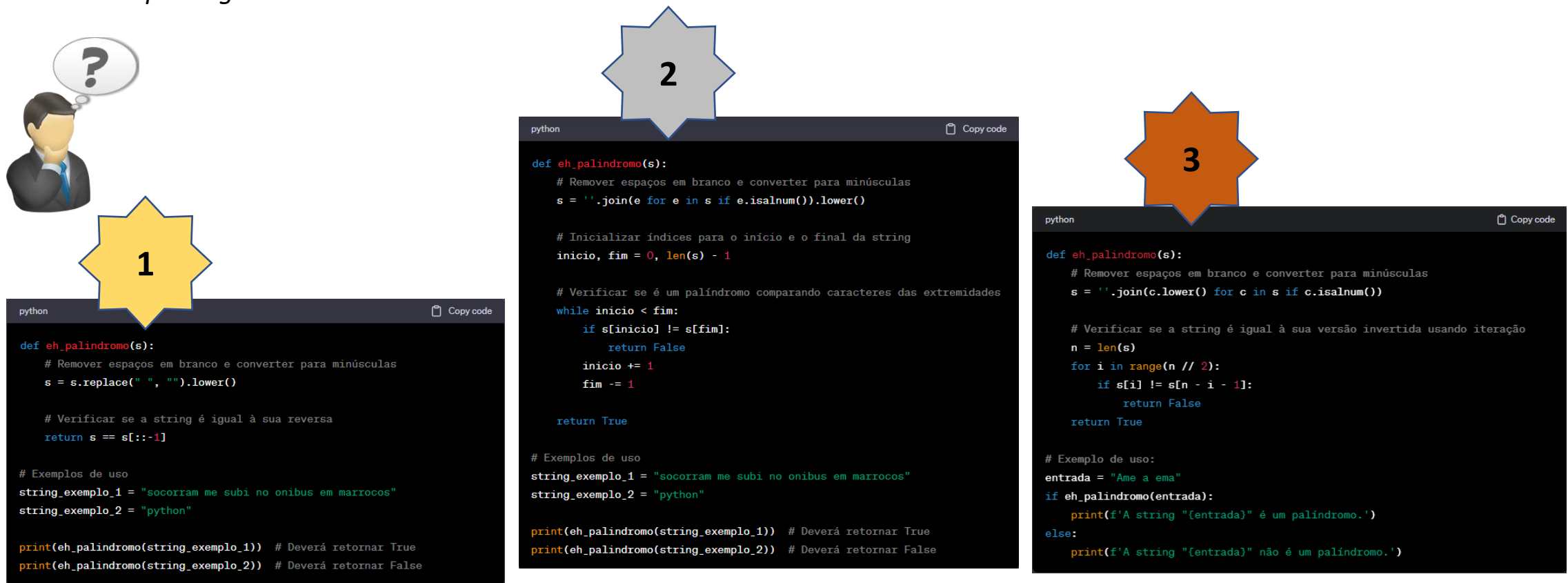


Figura 83 - Fonte: Adaptado de [29] pelo autor



No exemplo, o *Prompt* "Escreva uma função em Python que verifique se uma determinada *String* é um palíndromo" foi passado três vezes, e três respostas diferentes foram geradas (Figura 84 a, b, c). Aqui, estamos usando como referência o ChatGPT, que utiliza um modelo já treinado (no caso, o GPT 3.5), apenas para ilustrar que no Estágio 3 de treinamento um analista humano faria uma comparação e um ranqueamento das respostas geradas.

*Note que pode ser difícil fazer estas comparações, e mesmo especialistas podem demorar horas para julgar qual é a melhor resposta gerada.*



The diagram illustrates three different Python code snippets for a palindrome checker function, labeled 1, 2, and 3. A person with a question mark icon is shown to the left of the snippets, indicating a comparison or evaluation process.

**Snippet 1:**

```
python
Copy code

def eh_palindromo(s):
    # Remover espaços em branco e converter para minúsculas
    s = s.replace(" ", "").lower()

    # Verificar se a string é igual à sua reversa
    return s == s[::-1]

# Exemplos de uso
string_exemplo_1 = "socorram me subi no onibus em marrocos"
string_exemplo_2 = "python"

print(eh_palindromo(string_exemplo_1)) # Deverá retornar True
print(eh_palindromo(string_exemplo_2)) # Deverá retornar False
```

**Snippet 2:**

```
python
Copy code

def eh_palindromo(s):
    # Remover espaços em branco e converter para minúsculas
    s = ''.join(e for e in s if e.isalnum()).lower()

    # Inicializar índices para o início e o final da string
    inicio, fim = 0, len(s) - 1

    # Verificar se é um palíndromo comparando caracteres das extremidades
    while inicio < fim:
        if s[inicio] != s[fim]:
            return False
        inicio += 1
        fim -= 1

    return True

# Exemplos de uso
string_exemplo_1 = "socorram me subi no onibus em marrocos"
string_exemplo_2 = "python"

print(eh_palindromo(string_exemplo_1)) # Deverá retornar True
print(eh_palindromo(string_exemplo_2)) # Deverá retornar False
```

**Snippet 3:**

```
python
Copy code

def eh_palindromo(s):
    # Remover espaços em branco e converter para minúsculas
    s = ''.join(c.lower() for c in s if c.isalnum())

    # Verificar se a string é igual à sua versão invertida usando iteração
    n = len(s)
    for i in range(n // 2):
        if s[i] != s[n - i - 1]:
            return False
    return True

# Exemplo de uso:
entrada = "Ame a ema"
if eh_palindromo(entrada):
    print(f'A string "{entrada}" é um palíndromo.')
else:
    print(f'A string "{entrada}" não é um palíndromo.')
```

Figura 84 (a, b, c)

Feitos os ranqueamentos das respostas, temos dois tipos de *tokens* na matriz (B, T), o *Batch de Treinamento*:

- Os *tokens* da frase com a pergunta (*Prompt*) (em azul), que são idênticos nas três linhas da tabela,
- Os *tokens* com as respostas geradas pelo Modelo STF (em amarelo), diferentes em cada linha da tabela.

No exemplo da Figura 85 foram obtidas três respostas.

Ao final de cada resposta, é introduzido na matriz (B,T) do *Batch de Treinamento* um novo *token* (em verde), que representa um "valor de recompensa" (<reward>) calculado pelo Modelo de Recompensas (RM) que tem a predição sobre "o quão boa foi aquela resposta para o *Prompt* original".

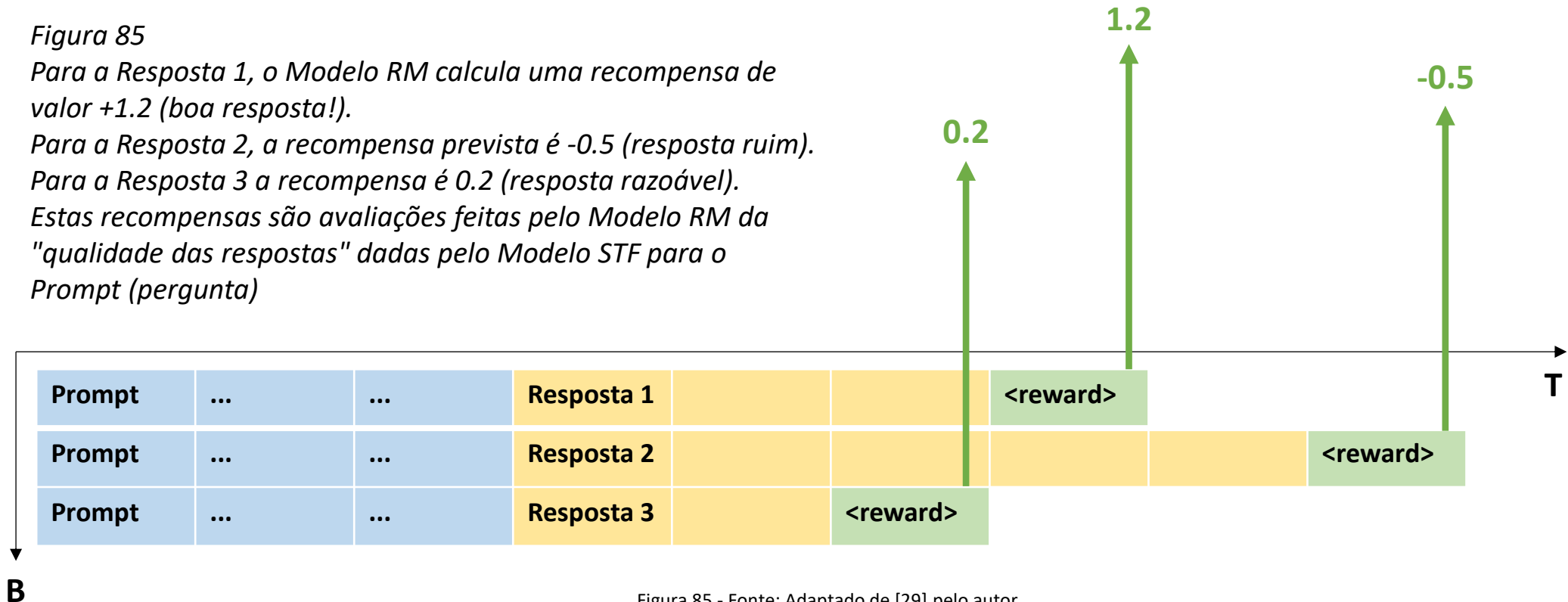
Figura 85

Para a Resposta 1, o Modelo RM calcula uma recompensa de valor +1.2 (boa resposta!).

Para a Resposta 2, a recompensa prevista é -0.5 (resposta ruim).

Para a Resposta 3 a recompensa é 0.2 (resposta razoável).

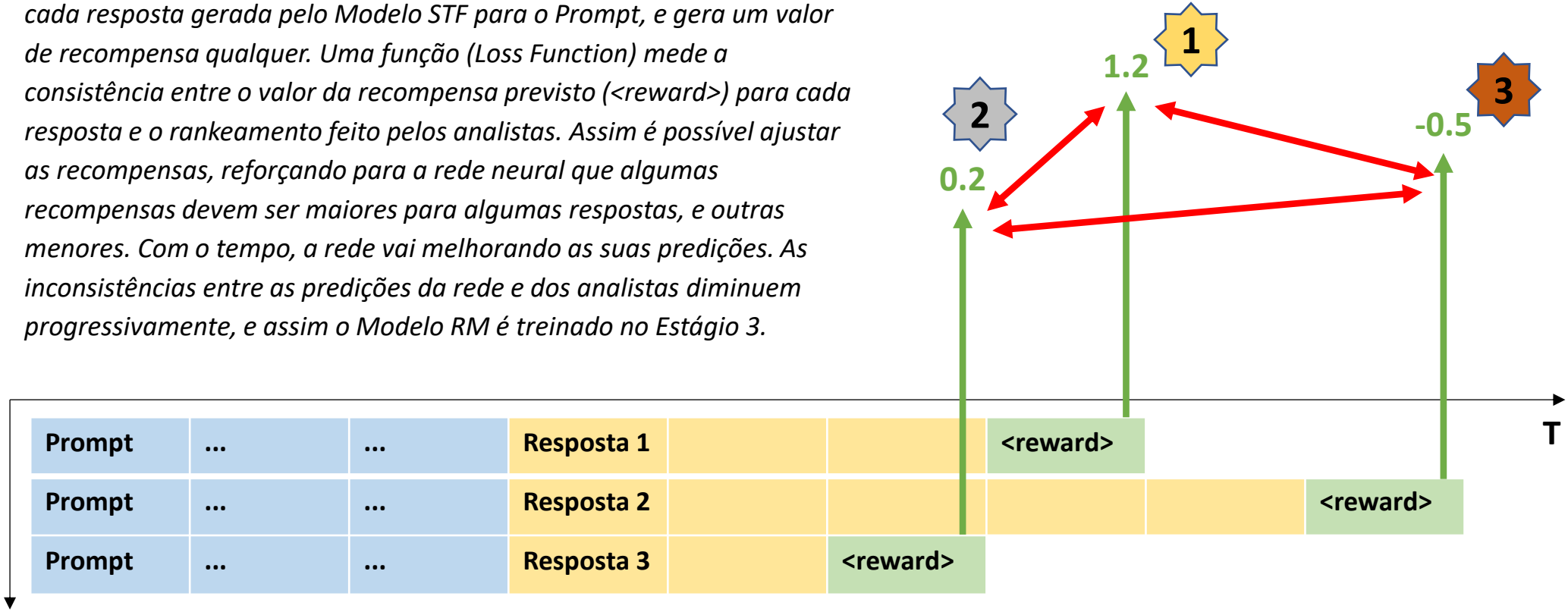
Estas recompensas são avaliações feitas pelo Modelo RM da "qualidade das respostas" dadas pelo Modelo STF para o *Prompt* (pergunta)



*Pergunta: Como saber se as avaliações feitas pelo Modelo RM são boas?*

*Resposta: Nós já sabemos qual é a melhor resposta (ground truth), pois as respostas foram ranqueadas pelos analistas. Assim podemos avaliar a "consistência" das recompensas geradas pela rede. Para cada resposta, uma função de perda (Loss Function) calcula a consistência entre a recompensa estimada e o ranking feito pelos analistas humanos. Havendo diferenças, são feitos ajustes para melhorar as estimativas das recompensas.*

*Figura 86 - No início, o Modelo RM apenas "adivinha" o quão boa foi cada resposta gerada pelo Modelo STF para o Prompt, e gera um valor de recompensa qualquer. Uma função (Loss Function) mede a consistência entre o valor da recompensa previsto (<reward>) para cada resposta e o ranqueamento feito pelos analistas. Assim é possível ajustar as recompensas, reforçando para a rede neural que algumas recompensas devem ser maiores para algumas respostas, e outras menores. Com o tempo, a rede vai melhorando as suas previsões. As inconsistências entre as previsões da rede e dos analistas diminuem progressivamente, e assim o Modelo RM é treinado no Estágio 3.*



**B**

Figura 86 - Fonte: Adaptado de [29] pelo autor



## Estágio 4 - Aprendizado com Reforço

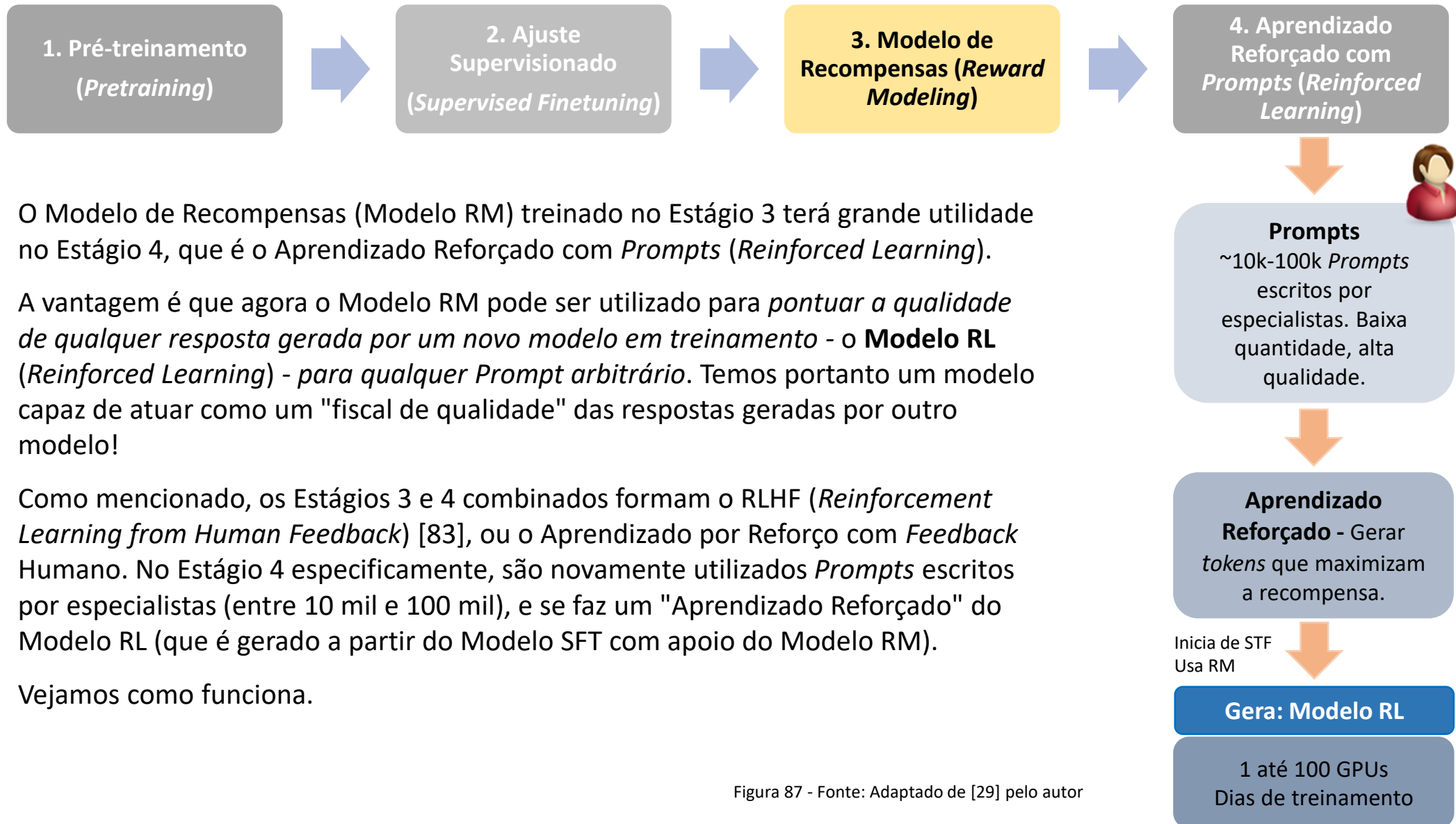


Figura 87 - Fonte: Adaptado de [29] pelo autor

- Os *tokens* da frase com a pergunta (*Prompt*) (em azul), que são idênticos nas três linhas da tabela.
- Os *tokens* com as respostas geradas pelo Modelo STF (em amarelo), diferentes em cada linha da tabela.
- Os *tokens* que representam "valores de recompensa" (<reward>) agora gerados pelo Modelo de Recompensas (RM), que agora está fixo, lembrando que estes *tokens* nos informam sobre a "qualidade" de cada resposta gerada. O objetivo no Estágio 4 é aumentar as probabilidades das melhores respostas, e reduzir as probabilidades das respostas ruins.

A diagram illustrating a sequence of actions and rewards. A horizontal timeline is shown with four segments. The first segment is orange and has a green arrow pointing up to the value 0.2. The second segment is light green and contains the text <reward>. The third segment is orange. The fourth segment is light green and contains the text <reward>. A green arrow points up from the end of the timeline to the value -1.2. A green arrow points up from the boundary between the second and third segments to the value 1.0.



Ao final do Estágio 4 temos um Modelo RL (*Reinforced Learning*), ou Modelo RLHF, uma evolução do Modelo SFT, já capaz de interagir com usuários através de *Prompts* e gerar boas respostas, e portanto pode ser publicado como um "Assistente GPT".

Diz-se que "o ChatGPT é baseado no modelo RLHF" pois seus modelos (GPT 3.5, GPT 4.0) seguem o *Pipeline* completo que descrevemos. Porém, há outros modelos de linguagem que seguem *Pipelines* de treinamento diferentes. Por exemplo, em [29] Andrej Karpathy nos ensina que o modelo Vicuna (13 bilhões de parâmetros) é um Modelo SFT, ou seja, é treinado apenas com os dois primeiros Estágios.

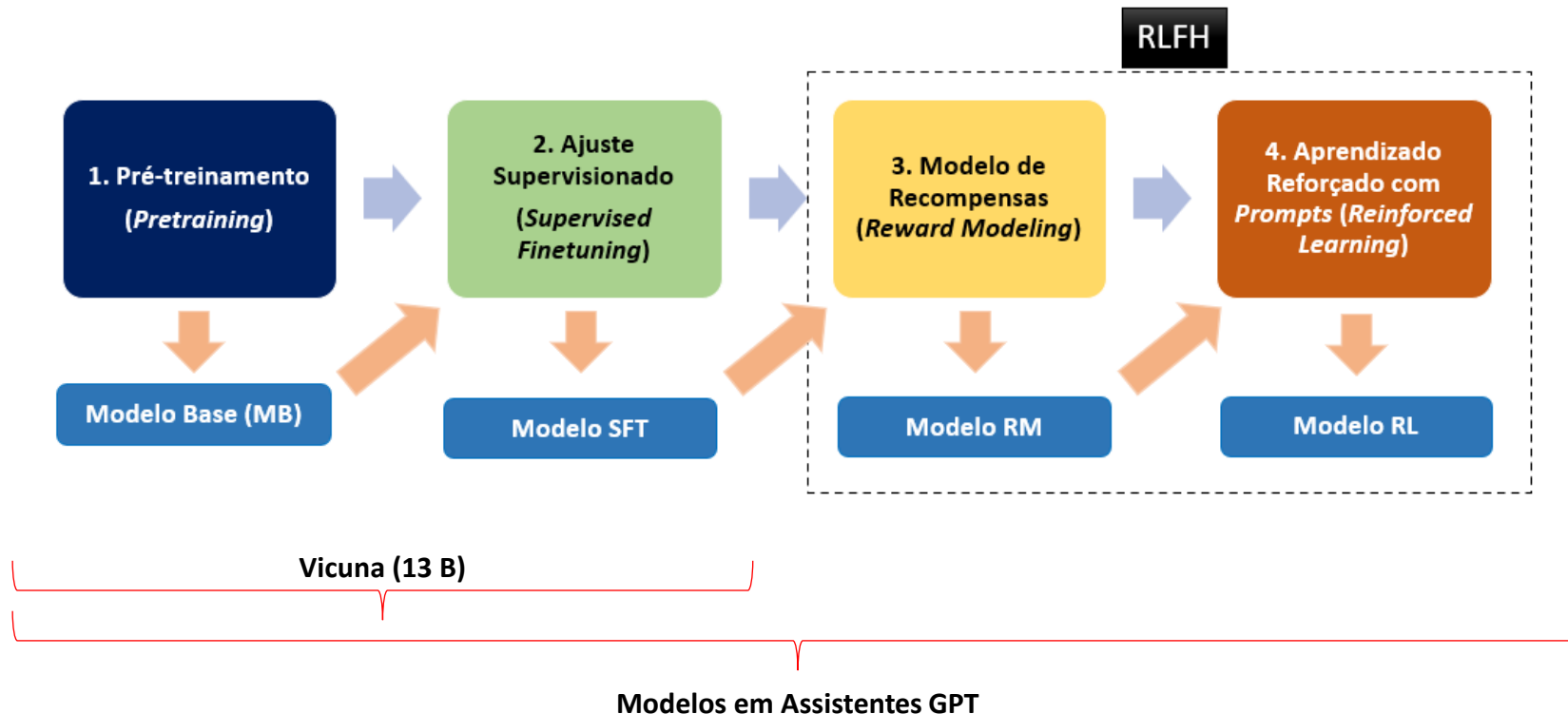


Figura 89 - Fonte: Adaptado de [29] pelo autor

## 3.2. Vantagens e limitações dos modelos RLHF

Vimos os quatro Estágios que formam o treinamento de Assistentes GPT, e destacamos a participação de humanos em todos os estágios com exceção do primeiro, onde o treinamento é não supervisionado. Vimos que os dois últimos estágios formam o RLHF (*Reinforcement Learning from Human Feedback*), que provou dar resultados melhores do que se o treinamento fosse interrompido ao final do Estágio 2.

Figura 90 - O gráfico mostra taxas de "Win rate" para diferentes modelos, que mostram a "preferência de seres humanos sobre textos gerados por diferentes modelos". Os dados mostram que humanos preferem tokens gerados por Modelos PPO (que são Modelos RLHF), em relação aos Modelos STF (que vão apenas até o Estágio 2 do Pipeline), e também em relação aos Modelos Base (MB) como o GPT no gráfico, que passou apenas pelo Estágio 1 do Pipeline. Os Modelos RLHF funcionam melhor. O gráfico é reproduzido em [29] a partir do artigo sobre o **InstructGPT** [84].

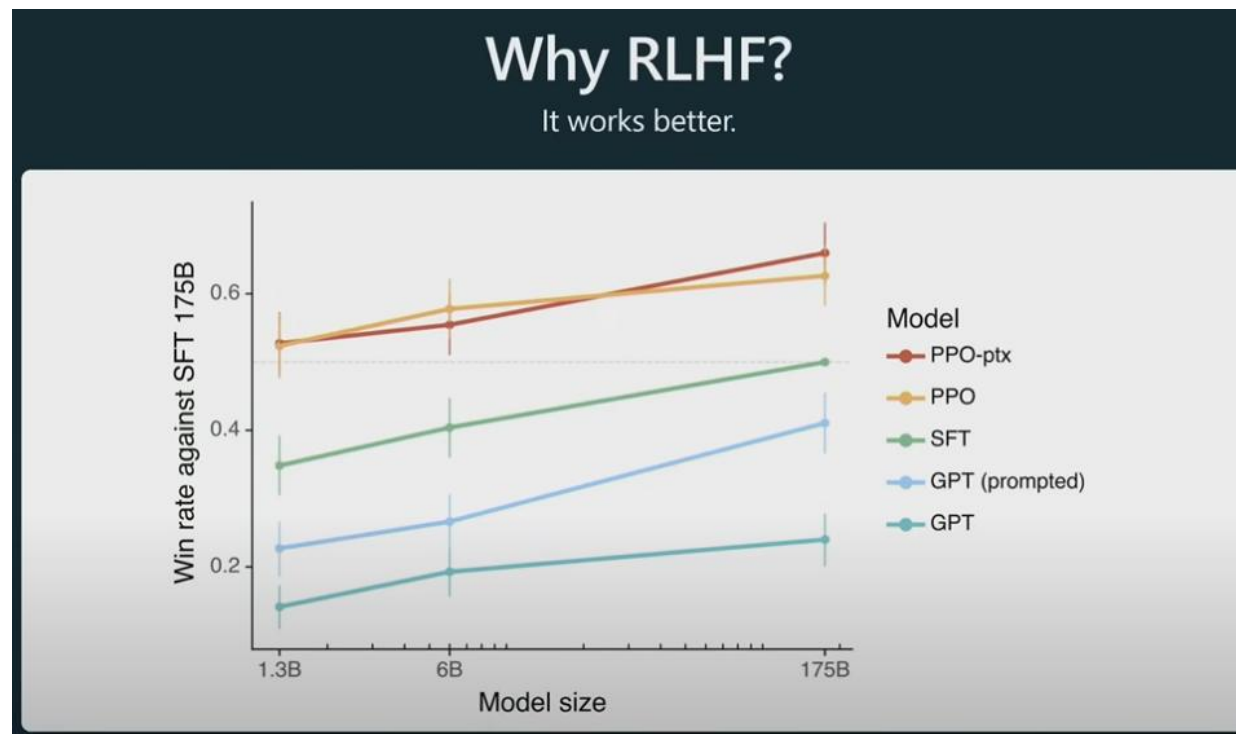


Figura 90 - Fonte: [29]

Vale destacar que há certas tarefas onde os Modelos Base (MB) podem gerar *melhores resultados* do que os Modelos RL (ou RLHF). Ocorre que os Modelos Base têm maior "entropia", um termo técnico que significa que podem gerar maior variedade de respostas. No exemplo de [29], a tarefa é "gerar mais nomes de Pokemon a partir de alguns exemplos fornecidos" - uma tarefa que requer maior capacidade de variação de respostas. Neste teste, o Modelo Base tente a se sair melhor que um Modelo RL. Ou seja, Modelos RLHF *não são sempre melhores que outros modelos* - depende da aplicação que se tem em mente.

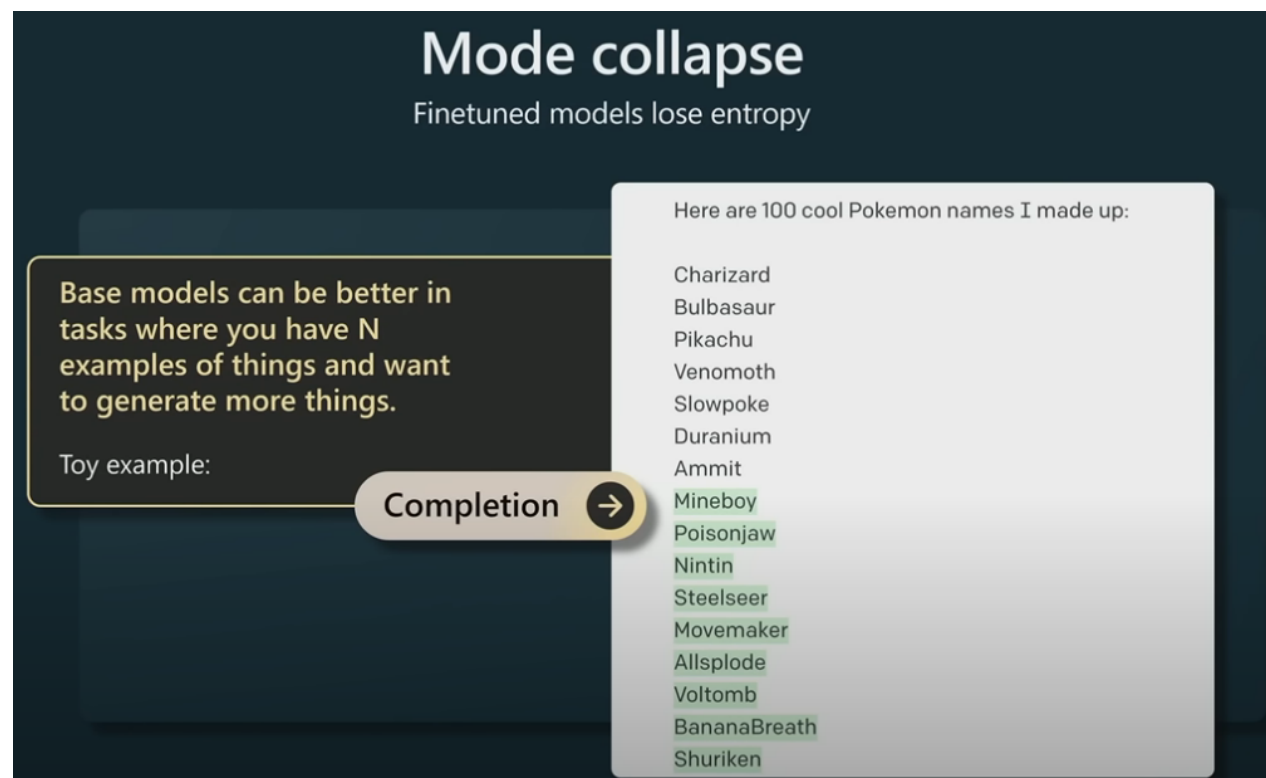


Figura 91 - Fonte: [29]

Para os interessados, há um *ranking* [85] comparando diversos *chatbots* (assistentes) baseados em diferentes modelos de linguagem, utilizando a métrica "*Elo Rating*". Alguns modelos são proprietários, outros são *open source* (Figura 92).

Em maio de 2023 o GPT-4 da OpenAI ocupava a liderança, seguido pelo Claude da Anthropic, e pelo GPT-3.5 Turbo da OpenAI.

*Os três primeiros modelos no comparativo são modelos RLHF.*




Rank	Model	Elo Rating	Description	License
1	 <a href="#">GPT-4</a>	1274	ChatGPT-4 by OpenAI	Proprietary
2	 <a href="#">Claude-v1</a>	1224	Claude by Anthropic	Proprietary
3	 <a href="#">GPT-3.5-turbo</a>	1155	ChatGPT-3.5 by OpenAI	Proprietary
4	<a href="#">Vicuna-13B</a>	1083	a chat assistant fine-tuned from LLaMA on user-shared conversations by LMSYS	Weights available; Non-commercial
5	<a href="#">Koala-13B</a>	1022	a dialogue model for academic research by BAIR	Weights available; Non-commercial
6	<a href="#">RWKV-4-Raven-14B</a>	989	an RNN with transformer-level LLM performance	Apache 2.0
7	<a href="#">Oasst-Pythia-12B</a>	928	an Open Assistant for everyone by LAION	Apache 2.0
8	<a href="#">ChatGLM-6B</a>	918	an open bilingual dialogue language model by Tsinghua University	Weights available; Non-commercial
9	<a href="#">StableLM-Tuned-Alpha-7B</a>	906	Stability AI language models	CC-BY-NC-SA-4.0
10	<a href="#">Alpaca-13B</a>	904	a model fine-tuned from LLaMA on instruction-following demonstrations by Stanford	Weights available; Non-commercial
11	<a href="#">FastChat-T5-3B</a>	902	a chat assistant fine-tuned from FLAN-T5 by LMSYS	Apache 2.0
12	<a href="#">Dolly-V2-12B</a>	863	an instruction-tuned open large language model by Databricks	MIT
13	<a href="#">LLaMA-13B</a>	826	open and efficient foundation language models by Meta	Weights available; Non-commercial

Figura 92 - Fonte: Adaptado de [85] pelo autor

### 3.3. Se você quer respostas mais precisas, peça por isso.

Humanos, quanto pensam em um problema, mantêm um monólogo interno e silencioso na cabeça. *"Será que é isso mesmo? Eu não entendo deste assunto, não sou bom em Geografia, vou verificar esta informação...."*.

Um alerta feito por Andrej Karpathy em [29] é que os grandes modelos de linguagem não "pensam" como humanos, não mantêm um "monólogo interior" enquanto resolvem problemas, *não sabem o que não sabem*, não sabem no que são bons ou não, não fazem reflexões, não executam "testes de sanidade" para verificar o próprio raciocínio. O que os modelos de linguagem (autorregressivos) fazem - como já sabemos - é **gerar o próximo token em uma sequência**.

*E são muito eficientes nisso.*

Contam com imenso *knowledge base* de fatos sobre diferentes áreas do conhecimento que foi utilizado no Estágio 1 do seu treinamento, e além disso possuem uma imensa e eficiente "memória" para dados que é a sua *janela de contexto*, sempre disponível para o mecanismo de Atenção na arquitetura *Transformer*.

- All of the internal monologue is stripped away in the text LLMs train on
- They spend the ~same amount of compute on every token
- => **LLMs don't reproduce this behavior by default!**
- They don't know what they don't know, they imitate the next token
- They don't know what they are good at or not, they imitate the next token
- They don't reflect. They don't sanity check. They don't correct their mistakes along the way
- They don't have a separate "inner monologue stream in their head"
- They do have very large fact-based knowledge across a vast number of areas
- They do have a large and ~perfect "working memory" (their context window)



Os grandes modelos de linguagem são treinados em *Datasets* que podem conter informações erradas. Em uma questão sobre Física, por exemplo, pode existir no *Dataset* uma resposta dada por um aluno, totalmente errada, e uma outra resposta dada por um ganhador do Prêmio Nobel, que é correta. A princípio, o modelo não sabe diferenciar, ambas fazem parte dos dados do treinamento. Mas o modelo é capaz de diferenciar "soluções de baixa qualidade" de "soluções de alta qualidade".

*Por incrível que pareça, se você quer mais precisão e melhor performance, você deve pedir por isso. Por exemplo, passar no Prompt a instrução "pense passo a passo, como se você fosse um grande especialista neste assunto, com um alto grau de inteligência para ter certeza que teremos a resposta correta" (Figura 94). É bem possível que você obtenha respostas melhores. Isso faz com que a rede *Transformer* funcione melhor, pois ela vai ignorar as "soluções de baixa qualidade" que por *default* não seriam ignoradas, já que tanto a resposta do aluno quanto a do especialista são parte da linguagem natural, e o modelo foi treinado para "imitar" a linguagem.*

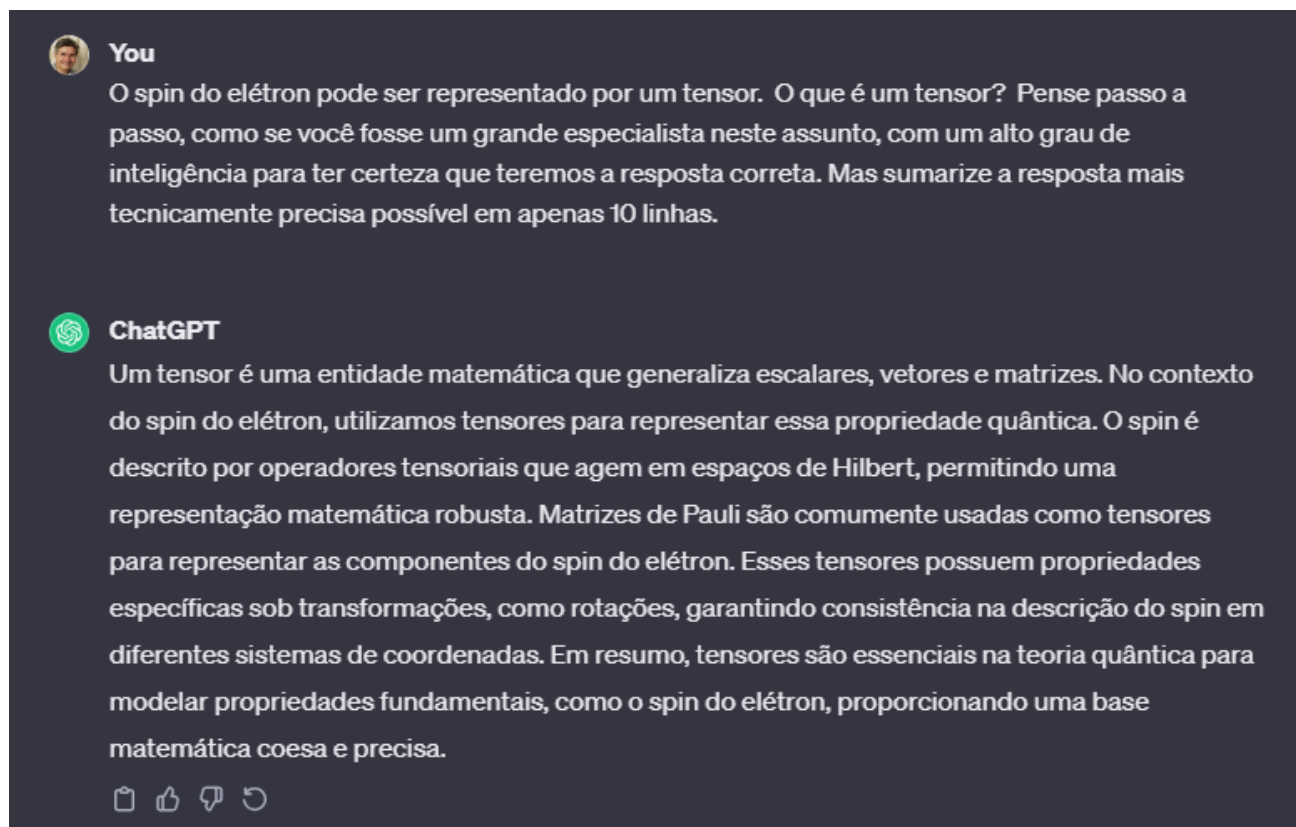


Figura 94



O tema da *Engenharia de Prompts* é vasto demais para ser abordado aqui [86, 87, 88], mas segue uma dica - em tarefas que envolvam cálculos matemáticos ou algum tipo de raciocínio lógico (*reasoning*), é conveniente instruir o modelo explicitamente no *Prompt* para pensar passo a passo para resolver o problema - assim, o modelo vai dividir o "raciocínio" em mais *tokens* e os resultados (respostas geradas) tendem a ser melhores. Esta abordagem é chamada *Chain-of-Thought Prompting* [89].

**You**

Uma caixa tem 16 bolas. Metade das bolas é de golfe. Metade das bolas de golfe é azul. Quantas bolas que não são azuis existem na caixa? Vamos pensar passo a passo.

**ChatGPT**

Vamos pensar passo a passo:

1. A caixa tem 16 bolas.
2. Metade das bolas é de golfe, o que significa que há  $16 / 2 = 8$  bolas de golfe.
3. Metade das bolas de golfe é azul, então há  $8 / 2 = 4$  bolas de golfe azuis.

Agora, queremos saber quantas bolas de golfe não são azuis. Se há 4 bolas de golfe azuis, então há  $8 - 4 = 4$  bolas de golfe que não são azuis.

Portanto, há 4 bolas de golfe que não são azuis na caixa.

Figura 95

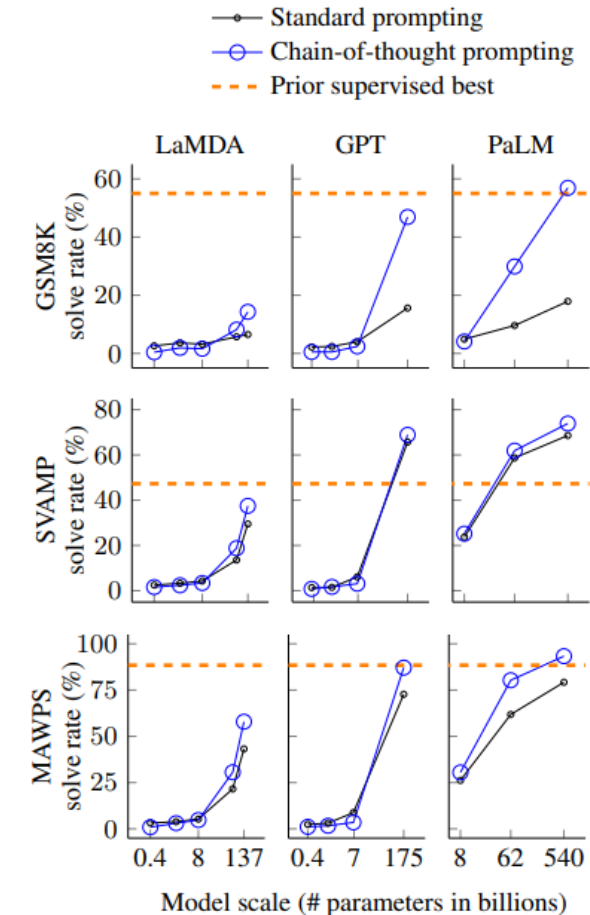


Figure 4: Chain-of-thought prompting enables large language models to solve challenging math problems. Notably, chain-of-thought reasoning is an emergent ability of increasing model scale. Prior best numbers are from Cobbe et al. (2021) for GSM8K, Jie et al. (2022) for SVAMP, and Lan et al. (2021) for MAWPS.

Figura 96 - Fonte: [89]

Exemplo de *Chain of Thought Prompting* retirado de [89].

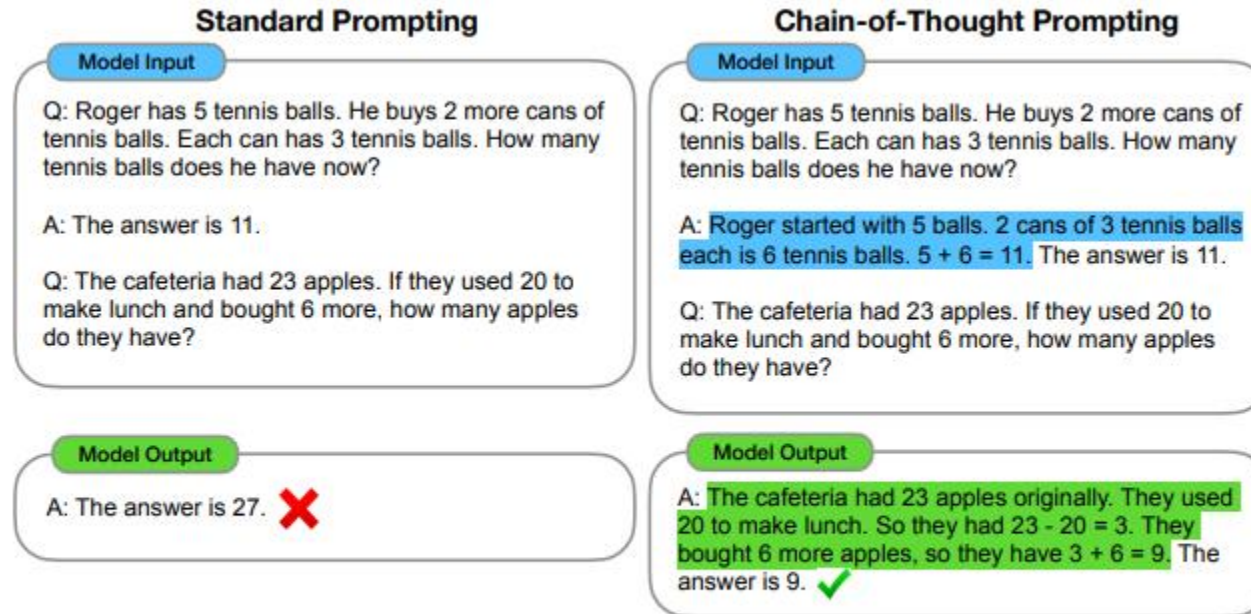


Figure 1: Chain-of-thought prompting enables large language models to tackle complex arithmetic, commonsense, and symbolic reasoning tasks. Chain-of-thought reasoning processes are highlighted.

Figura 97 - Fonte: [89]

A Figura 97 mostra experimentos com três grandes modelos de linguagem que mostram que o uso de Chain-of-Thought Prompting pode melhorar significativamente a performance em tarefas que envolvem aritmética, senso comum e raciocínio lógico.

# Arithmetic Reasoning on GSM8K

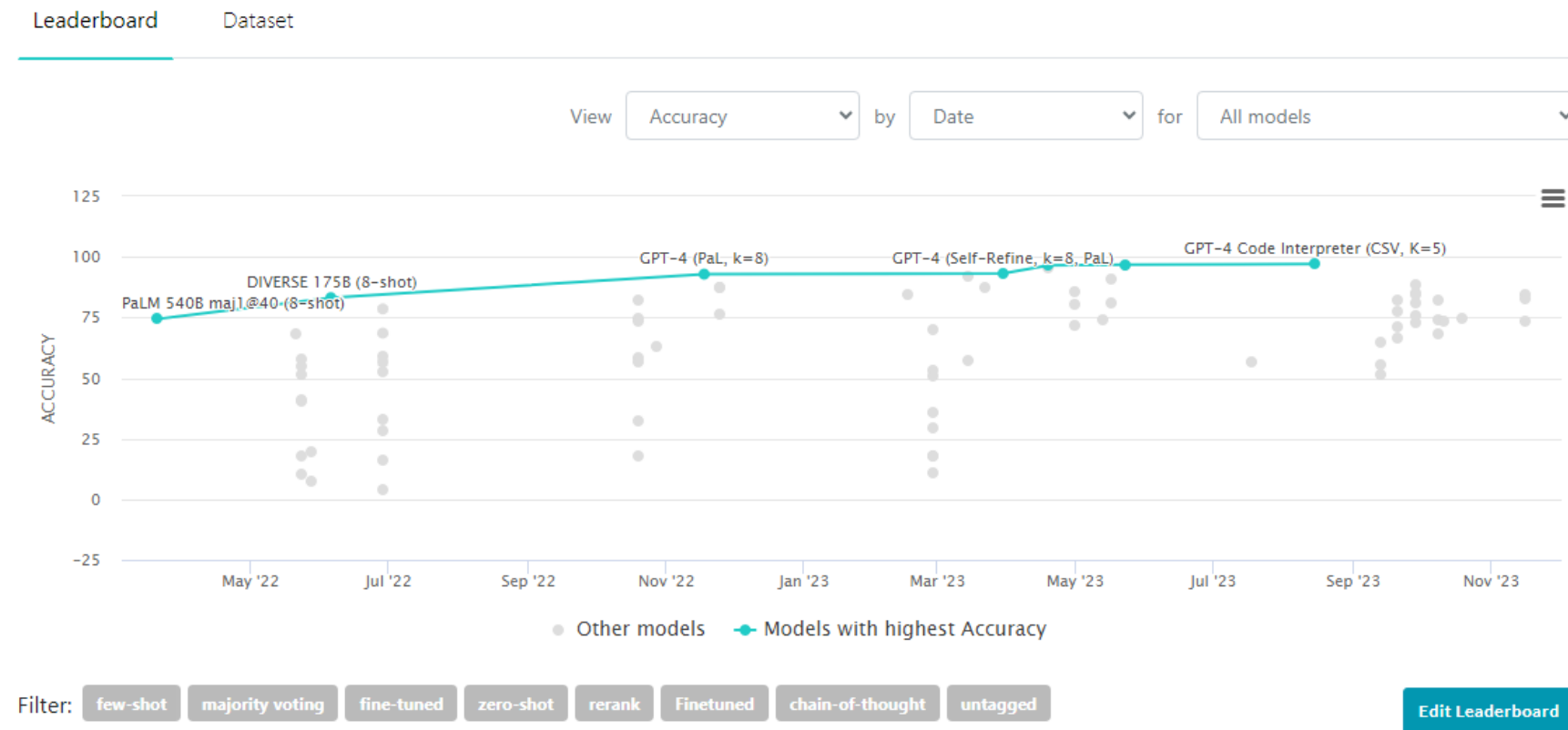


Figura 98 - Fonte: [90]

Figura 98 - Para o modelo PaLM com 540 bilhões de parâmetros, o uso de apenas oito exemplos com Chain-of-Thought levou o modelo a alcançar o estado da arte em acurácia no comparativo GSM8K de raciocínio aritmético, ultrapassando o modelo GPT-3. O modelo PaLM foi depois superado neste mesmo benchmark pelo GPT-4 da OpenAI, que ocupa a primeira posição neste ranking no momento (ver [90]).

### 3.4. Para tarefas mais especializadas, use *plugins*

Você também pode desonerar o seu assistente GPT de executar tarefas nas quais ele não é bom, fazendo a integração com *plugins* que são aplicativos especializados. Os *plugins* também permitem que o ChatGPT acesse serviços de terceiros. Também é possível criar novos *plugins* seguindo a documentação técnica da OpenAI [91]. Como explicado por Karpathy em [29], *o modelo não sabe o que ele não sabe, ou no que ele não é bom*. Então você pode dizer ao ChatGPT algo como "você não é muito bom em aritmética, então utilize esta calculadora para fazer estes cálculos".

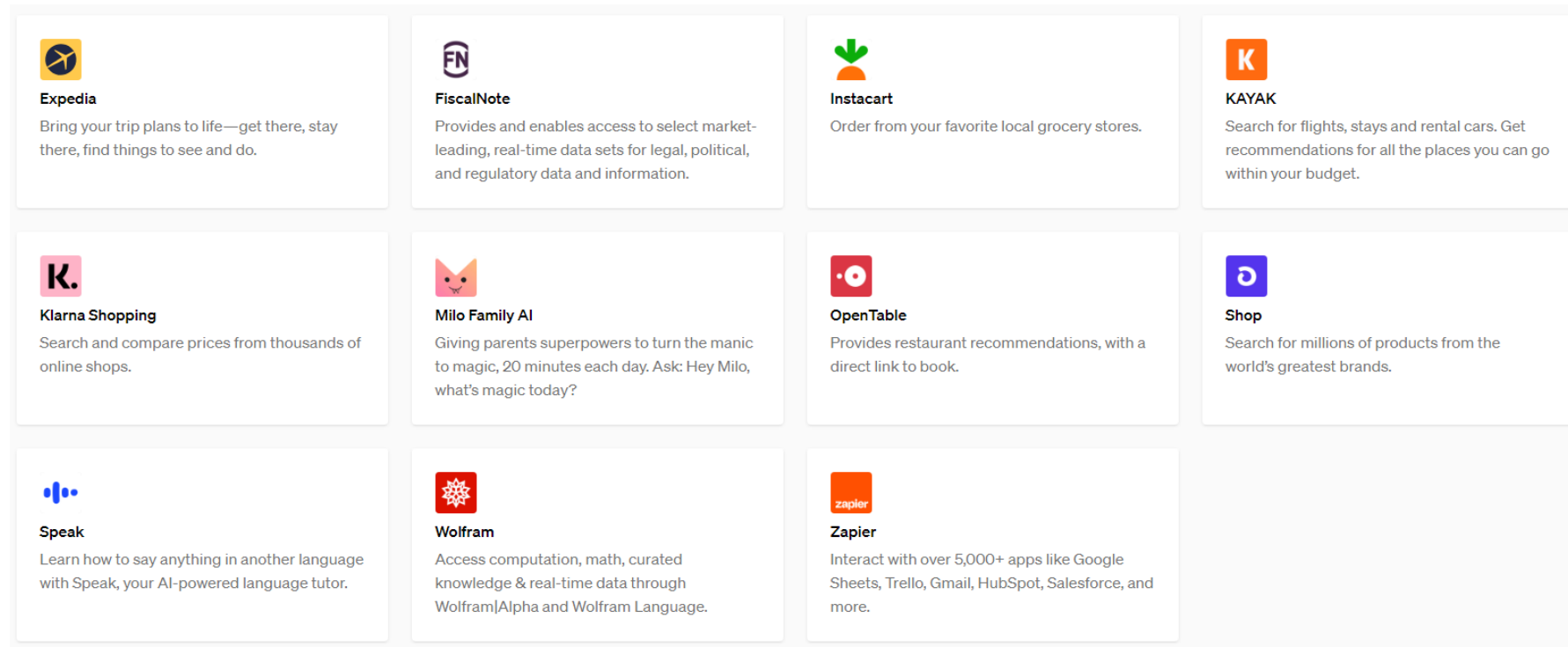


Figura 99 - Fonte: [91]

# Capítulo 4 - *Atenção é tudo o que você precisa*

Neste Capítulo, vamos abordar de forma introdutória algumas das inovações mais importantes no contexto da arquitetura para NLP (*Natural Language Processing*) a partir de 2019 - os **mecanismos de Atenção** e os modelos *Transformer*.

Estas técnicas, antes restritas apenas para especialistas, estão rapidamente se tornando mais acessíveis com a maior facilidade de utilizar redes neurais na nuvem capazes de processar quantidades massivas de dados (por exemplo, através dos serviços da Microsoft Azure OpenAI). Por conta disso, já estão sendo empregadas em aplicações de negócio por organizações em todo o mundo. Ou seja, não são mais curiosidades acadêmicas - são *importantes diferenciais de negócios* em um mercado cada vez mais impactado pela inteligência artificial.

Antes de abordar as redes *Transformer*, porém, vamos tratar brevemente (e de forma bem introdutória) das RNNs (Redes Neurais Recorrentes), arquiteturas importantes de redes neurais que surgiram nos anos 80.

Veremos que o *Transformer* é uma evolução da RNNs, eliminando sua principal restrição - enquanto as RNNs só podem ser processadas sequencialmente, as redes *Transformer* fazem *computações em paralelo*, graças aos seus mecanismos de Atenção.

Por convenção, neste Capítulo vamos utilizar o termo 'Atenção' capitalizado quando estivermos falando do mecanismo técnico utilizado nas redes *Transformer*, para diferenciar do termo 'atenção' usual da linguagem natural.

## 4.1. Redes Neurais

Antes de discutir as sofisticadas redes neurais *Transformer* precisamos ter pelo menos uma noção básica sobre o que é uma rede neural. O assunto é bastante técnico e vamos abordá-lo de modo superficial - caso você queira se aprofundar, encontrará informação de qualidade em [43, 44, 92 e 93].

Assim como na biologia, uma rede neural como as utilizadas na Inteligência Artificial e aprendizado profundo (*Deep Learning*) é uma coleção de "neurônios" conectados, geralmente agrupados em camadas. Aqui, cada "neurônio" pode ser modelado como uma função matemática simples, que recebe uma ou mais entradas (*inputs*), aplica pesos a estes *inputs*, faz uma soma, passa o resultado por uma função de ativação e gera uma saída (*output*).

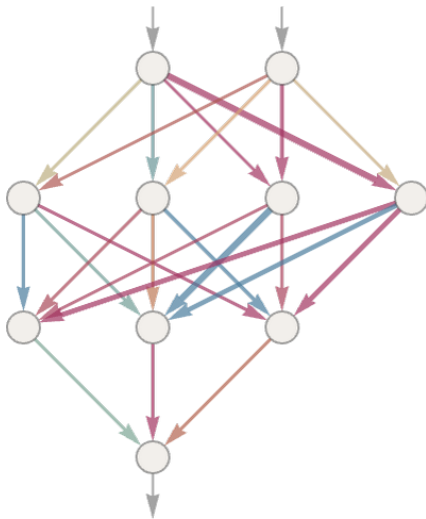


Figura 100 - Fonte: [43]

Figura 100 - Cada círculo é um neurônio, que recebe uma ou mais entradas e gera uma saída numérica. Cada entrada  $x$  (*input*) tem um peso  $w$  (*weight*) associado. Matematicamente, se  $x_1, x_2, \dots, x_n$  são os *inputs*, e  $w_1, w_2, \dots, w_n$  são os pesos correspondentes para cada *input*, o cálculo do resultado gerado em cada neurônio (*output*) pode ser expresso como:

$$\text{output} = \text{função\_ativação} (w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n + \text{bias}) \quad [4.1]$$

Onde o *bias* é um termo adicional que ajuda a ajustar a saída do neurônio. A função de ativação introduz não linearidades na operação do neurônio, permitindo que a rede aprenda padrões complexos.

Para simplificar, digamos que cada neurônio avalia uma função numérica e gera um resultado. A rede neural é composta por várias camadas de neurônios, cada uma conectada à próxima. Cada camada pode ter um número diferente de neurônios, e cada neurônio pode se conectar a um ou mais neurônios na camada seguinte.

Para utilizar uma rede neural, precisamos alimentá-la com números no seu início, ou seja, na entrada da rede. Daí por diante, os neurônios em cada camada avaliam suas funções, passam os resultados para neurônios na próxima camada, e assim por diante, até que se chega em um resultado final, que é entregue na saída da rede. No exemplo de [43] (Figura 101), os números 0.5 e -0.8 entram na rede, vários processamentos são feitos nos diversos neurônios de cada camada, e ao final é gerado um valor (no caso, -1).

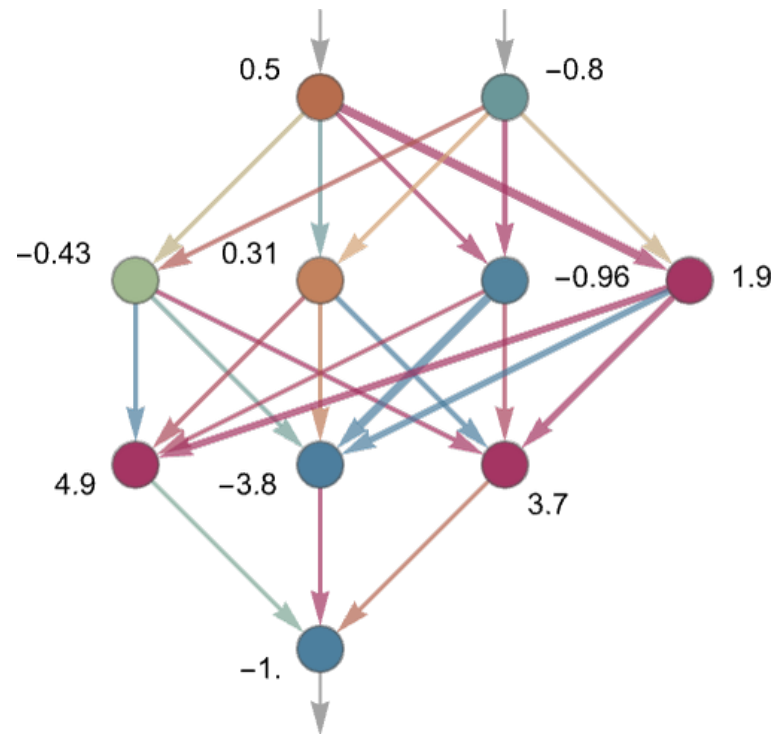


Figura 101 - Fonte: [43]



Assim como existem conexões (sinapses) entre os nossos neurônios biológicos que podem ter "intensidades" diferentes, as conexões entre os neurônios também possuem pesos, que podem ser números positivos ou negativos.

Ou seja, além dos pesos  $w_j$  que ponderam as entradas  $x_j$  em um neurônio específico em uma camada, as conexões entre neurônios de diferentes camadas em uma rede neural também são ponderadas por pesos. Esses pesos representam a força ou a importância da conexão entre os neurônios.

Se considerarmos uma rede neural com várias camadas, podemos denotar os pesos **entre os neurônios** da camada  $i$  e a camada seguinte  $i + 1$  como  $w_{ij}^{(l)}$ , onde  $l$  (de *layer*) é o índice da camada. Cada peso  $w_{ij}^{(l)}$  representa a influência da ativação do neurônio  $j$  na camada  $l$  na ativação do neurônio  $i$  na camada seguinte  $l + 1$ . Esses pesos podem ser positivos ou negativos, determinando se a conexão é excitatória (aumenta a ativação) ou inibitória (diminui a ativação).

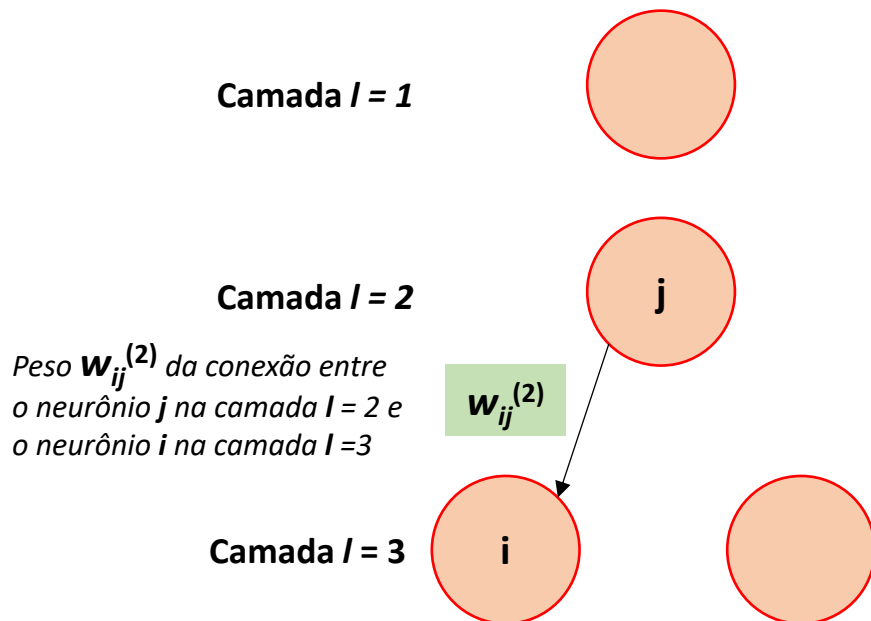


Figura 102

*Figura 102 - Quando falamos sobre a "influência da ativação de um neurônio  $j$  da camada  $l$  na ativação de um neurônio  $i$  da camada  $l + 1$ , estamos nos referindo ao fato de que a ativação do neurônio  $j$  será levada em conta no cálculo da entrada ponderada do neurônio  $i$  na camada  $l + 1$ . Em termos mais simples, a ativação de um neurônio em uma camada pode afetar a ativação de neurônios na próxima camada através das conexões ponderadas entre eles. Se a conexão entre  $j$  e  $i$  tem um **peso positivo** (excitatória), a ativação de  $j$  aumentará a entrada ponderada para  $i$ , potencialmente aumentando a ativação de  $i$  na camada  $l + 1$ . Se a conexão tem um **peso negativo** (inibitória), a ativação de  $j$  pode inibir a ativação de  $i$ .*



Este é portanto o papel do peso  $\mathbf{W}_{ij}$  (capitalizado) da conexão entre dois neurônios. Essa propagação de ativação através das camadas é fundamental para o funcionamento de uma rede neural, permitindo que ela aprenda representações complexas dos dados à medida que os pesos são ajustados durante o treinamento.

De onde vêm os pesos  $\mathbf{w}_j$  que ponderam as entradas  $\mathbf{x}_j$  na função de ativação [equação 4.1] de cada neurônio?

Os pesos  $\mathbf{w}_{ij}$  que são utilizados na função de ativação de um neurônio qualquer são **parâmetros da rede neural** e são **aprendidos durante o treinamento**. Para cada tipo de tarefa que desejamos que a rede neural execute existem diferentes escolhas para os pesos. Por exemplo, a tradução de textos envolve a compreensão da semântica e sintaxe em dois idiomas diferentes, de modo que os pesos da rede precisam capturar relações complexas entre palavras em diferentes línguas. Para isso, as redes neurais que treinam os LLMs têm muitas camadas e bilhões de pesos (além de vieses ou *bias*) que podem ser ajustados. Como vimos, o modelo GPT-3 tem 175 bilhões de parâmetros.

Inicialmente, esses pesos são atribuídos com valores aleatórios, e ao longo do processo de treinamento, são ajustados iterativamente para minimizar uma **função de perda**. O processo de treinamento geralmente utiliza a retropropagação de um erro (*backpropagation*), onde o erro entre as previsões feitas pela rede e os rótulos reais é *propagado de volta pela rede*. Os pesos  $\mathbf{w}_j$  nos neurônios são ajustados para reduzir esse erro. Isso é feito usando **algoritmos de otimização** como o **Gradiente Descendente** [94]. Durante cada iteração do treinamento, os pesos são ajustados de forma proporcional ao gradiente da função de perda em relação a esses pesos (na direção que reduz o valor da função de perda, ou seja, minimiza os erros).

Assim, os pesos  $\mathbf{w}_{ij}$  não são definidos *a priori*, mas são adaptados pela rede neural ao longo do treinamento para otimizar o desempenho da rede na tarefa específica para a qual está sendo treinada. Esse processo de aprendizado é uma das características fundamentais que permitem que as redes neurais aprendam a representar padrões complexos que existem nos dados (é o "*Learning*" do *Machine Learning*, ou mais precisamente, do "*Deep Learning*").

## 4.2. Redes Neurais Recorrentes (RNN - *Recurrent Neural Networks*)

A Seção 4.1 apresentou uma introdução às redes neurais "tradicionais". Aqui, vamos tratar das redes neurais recorrentes ou RNNs (*Recurrent neural networks*).

Uma Rede Neural Recorrente (RNN) é um tipo de arquitetura de rede neural projetada para lidar com dados sequenciais ou temporais. Ao contrário de redes neurais convencionais, as RNNs têm a capacidade de manter uma "memória" de eventos anteriores na sequência de dados de entrada. Isso as torna especialmente úteis para tarefas em que a ordem e a dependência temporal dos dados são importantes.

As RNNs podem trabalhar com qualquer tipo de dado sequencial, o que inclui textos e também áudio, vídeo ou mesmo códigos de computador. Há diferentes tipos de RNNs, incluindo as que têm arquitetura ENCODER - DECODER que são adequadas para tarefas de tradução por exemplo (tanto o ENCODER quanto o DECODER são compostos por redes neurais com várias camadas). *Assim, a RNN é uma arquitetura formada por várias redes neurais.*

A característica principal das RNNs é a presença de *loops* em sua estrutura, permitindo que informações sejam persistentemente transmitidas de um passo no tempo para o próximo. Isso é particularmente útil em tarefas como processamento de linguagem natural (PLN), previsão de séries temporais, tradução automática e qualquer tarefa que envolva dados sequenciais.

A estrutura básica de uma RNN consiste em um conjunto de unidades (neurônios) interconectadas, onde cada unidade recebe uma entrada e uma "memória" (ou estado oculto) da unidade anterior, além de uma entrada atual. A saída da unidade também é usada como entrada para a próxima unidade na sequência. Esse processo de recorrência permite que a rede mantenha informações sobre padrões em dados sequenciais.

Não vamos nos aprofundar nos detalhes técnicos das RNN, nosso propósito é apenas dar uma noção da arquitetura. Explicações detalhadas podem ser encontrada na vasta literatura disponível sobre o assunto [por exemplo, em 44, 92, 95, 96, 97, 98 e 99]. O diagrama de [96] mostra um exemplo de bloco ENCODER em uma RNN.

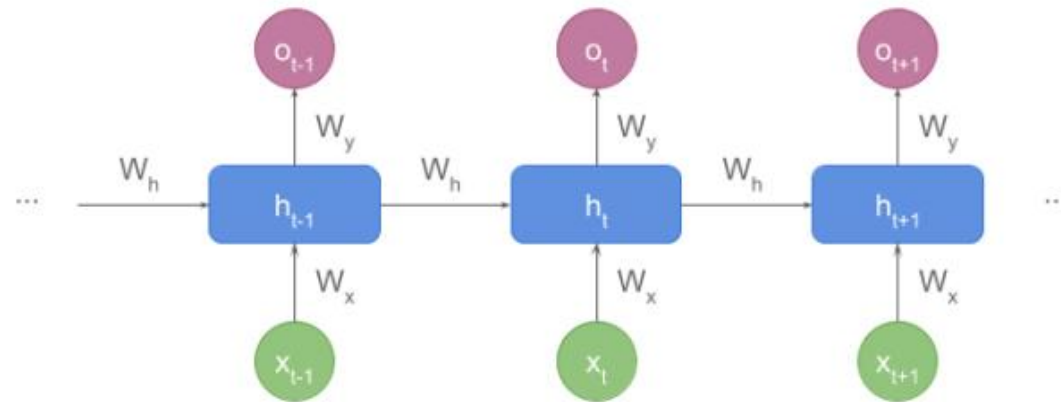


Figura 103 - Fonte: [96]

Cada bloco da rede é composto pelos seguintes elementos, no instante  $t$ :

#### ENTRADAS DO BLOCO

- **Vetor de Embeddings ( $x_t$ )**
- Vetor  $h_{t-1}$  **de estado oculto** (*hidden state vector*), contendo o estado imediatamente anterior ao bloco atual)

#### SAÍDAS DO BLOCO

- Vetor de saída  $o_t$  (output vector), *que nem sempre é gerado em todos os blocos*.
- Pesos da rede neural
- $W_x$  (pesos entre o o **vetor  $x_t$**  e o **vetor  $h_t$** )
- $W_h$  (pesos entre o **vetor  $h_{t+1}$**  e o **vetor  $h_t$** )
- $W_y$  (pesos entre o **vetor  $h_t$**  e o **vetor  $o_t$** )

Utilizando a partir de agora uma representação mais simples, o exemplo de [96] usa uma tarefa de tradução para mostrar o funcionamento de uma RNN.

A frase "Eu amo aprender" em Português será traduzida para "I love learning" (Inglês).

O ENCODER ou codificador é o bloco responsável por processar a sequência de entrada ou INPUT (*sequências de de tokens*), e codificá-la como um vetor numérico (C = Vetor de Contexto).

No diagrama simplificado, os  $H_t$  no ENCODER representam **vetores de estado** (sequências de estados ocultos ou *hidden states*). Cada vetor em  $t$  é atualizado como uma função do estado prévio  $H_{t-1}$  e do INPUT (**vetor de *Embedding*  $x_t$** ) fornecido em cada posição  $t$ . Em favor da simplicidade, foram omitidos os pesos  $Wx$ ,  $Wh$  e  $Wy$ .

O vetor C contém uma representação vetorial (vetor com números reais) da sequência completa que se quer traduzir ("Eu amo aprender") para o inglês.

### ENCODER

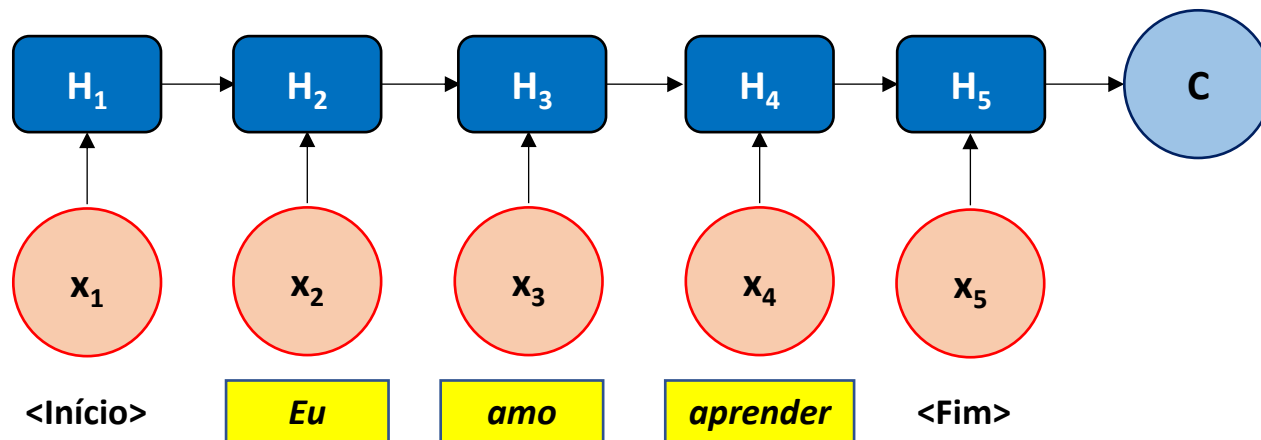


Figura 104 - Fonte: Adaptado de [96] pelo autor

O DECODER recebe e processa o Vetor de Contexto C contendo a representação final da sequência de entrada ("Eu amo aprender"). No DECODER, é iniciada a decodificação, que no caso é uma tradução, *token por token*. Uma função **softmax** procura o *token* no vocabulário com maior probabilidade, e é gerada a tradução da primeira palavra, que é realimentada na próxima camada do DECODER para ajudar na tradução da segunda palavra, "love", que é realimentada na rede, e assim, *token por token*, é gerada a **sequência de saída** ou OUTPUT ("I", depois "love", depois "learning").

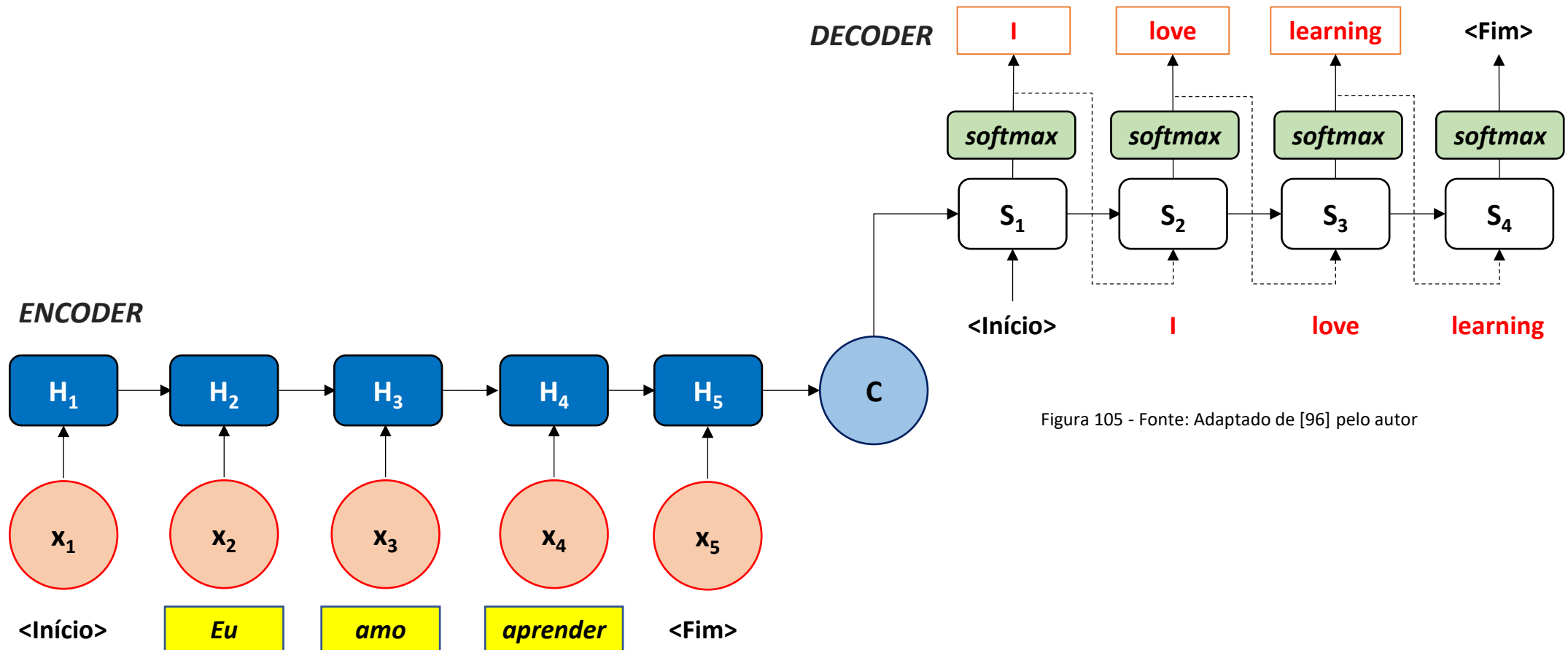


Figura 105 - Fonte: Adaptado de [96] pelo autor

As RNNs possuem vantagens e desvantagens [96]:

- Dentre as vantagens, as RNNs aceitam *Inputs* de qualquer tamanho (representações de tamanho variável) sem que isso afete o tamanho do modelo, e são adequadas para modelar eventos sequenciais.
- Dentre as desvantagens, as RNNs são lentas, pois *precisam ser processadas sequencialmente* (é necessário obter o estado anterior para calcular o estado atual). Como mencionado, se a computação que ocorre nas redes recorrentes for alinhada no tempo, é gerada uma sequência de estados ocultos (*hidden states*)  $h_t$ , como uma função do estado oculto anterior  $h_{t-1}$  e da entrada (*input*) no instante  $t$ . Este processo sequencial impede o paralelismo durante o treinamento, e quando as sequências são muito grandes há restrições de memória e a performance é afetada.

### **O problema do gradiente**

Há um outro problema que ocorre no treinamento das RNNs, que tem relação com um cálculo do *gradiente*, um vetor utilizado para atualizar os pesos  $\mathbf{W}$  (*weights*) nas diferentes camadas da rede (*backpropagation*) para que ela possa reduzir seus erros e fazer melhores previsões no futuro (parte importante do processo de "aprendizado").

*Quando o gradiente é pequeno, ele tende a diminuir (desaparecer) exponencialmente com o número de camadas (Vanishing Gradients), e os pesos ficam cada vez menores. Quando a rede tem muitas camadas, os efeitos dos Inputs das camadas anteriores são levados cada vez menos em conta. É como se a rede só tivesse "memória de curto prazo", e assim a rede não consegue "aprender".*

*Inversamente, se o gradiente é grande, ele aumenta exponencialmente com o número de camadas, tornando os pesos cada vez maiores e o aprendizado muito instável.*

Como solução para o problema do gradiente, foram propostas versões mais especializadas das RNNs com unidades GRU (*Gated Recurrent Unit*) [97], introduzidas em 2014, ou LSTM (*Long Short Term Memory*) cujo desenvolvimento foi iniciado nos anos 1990 [98]. Estas são o que podemos chamar de RNNs modernas, em oposição às RNNs tradicionais. Estas "novas versões" das RNNs utilizam *Gates* (um *Gate* é uma rede neural que controla o fluxo de informação de outra rede neural). Os *Gates* são capazes de aprender quais *inputs* em uma sequência são mais importantes e armazená-los em uma "memória adicional". O uso de unidades GRU ou LSTM ajudou a preservar dependências importantes, reduzindo um pouco a perda de informações pelo problema do gradiente. Porém, o problema do "gradiente que desaparece" não é totalmente resolvido pelas LSTM. Para um comparativo das redes RNN, GRU e LSTM ver [99].

Em 1998 surgiram as Redes Neurais Convolucionais (*Convolutional Neural Networks* - CNN) [100], um tipo de Rede Neural *Feed Forward* (FFNN). Em certa medida, as CNNs ajudam a mitigar o problema dos "gradientes que desaparecem" (*Vanishing Gradients*), embora não eliminem completamente o problema. Devido à sua arquitetura específica, que inclui camadas de convolução, *pooling* e, muitas vezes, ativações não-lineares, as CNNs são particularmente eficazes em tarefas de Visão Computacional e processamento de imagens (classificar objetos em imagens por exemplo). Com o tempo, as CNNs foram adaptadas para outros tipos de tarefas como processamento de áudio e de linguagem natural.

A convolução é uma técnica que permite aplicar transformações lineares diferentes para cada posição relativa em uma sequência. As camadas de convolução nas CNNs têm a propriedade de compartilhar parâmetros, o que significa que os mesmos pesos são usados em várias regiões da imagem. Esse compartilhamento de parâmetros ajuda a reduzir o número total de parâmetros na rede, tornando o treinamento mais eficiente. Além disso, a aplicação de funções de ativação não-lineares, como a ReLU (*Rectified Linear Unit*) [101] também ajuda a evitar o desaparecimento dos gradientes, pois introduz não-linearidades nas ativações das unidades. Porém, em redes muito profundas (muitas camadas ocultas) o problema dos gradientes ainda pode surgir, mesmo com o uso de técnicas como ativações não-lineares e inicialização adequada de pesos.

Nos anos 2010 surgiram os **mecanismos de Atenção** para redes neurais recorrentes, que permitiam modelar as dependências entre *tokens* independentemente da distância entre eles, tanto nas sequências de entrada (*input*) quanto de saída (*output*). Vamos ver a Atenção em detalhes neste Capítulo, mas por hora podemos pensar na Atenção como uma *medida de similaridade* (proximidade semântica ou contextual) entre diferentes palavras em uma sequência.

Uma das primeiras formulações de mecanismos de Atenção em contextos recorrentes foi introduzida em 2014 por Dzmitry Bahdanau e colegas em [102]. Esse trabalho mostrou que o mecanismo de Atenção permite que a rede focalize com maior ênfase palavras específicas da sequência de entrada durante o processamento, em vez de processar *todas* as palavras da sequência de maneira uniforme. Porém, mesmo nestas redes "já atentas" o processamento ainda era sequencial, e para sequências muito longas persistia a baixa eficiência computacional.

Uma solução brilhante - tanto para o "problema do gradiente" quanto para a melhoria da eficiência computacional - veio com a rede neural com arquitetura *Transformer* [80] que é baseada *unicamente* em mecanismos de Atenção, eliminando as convoluções ou as recorrências típicas das RNNs. Como consequência, as tarefas de tradução de idiomas por redes neurais (e outras) atingiram o estado da arte.

Enquanto nas RNNs a Atenção é aplicada de forma sequencial, nas redes *Transformer* a Atenção pode ser aplicada *em paralelo*, em várias posições da sequência de entrada simultaneamente. Este paralelismo na computação *reduziu o tempo e o custo do treinamento de grandes modelos de linguagem* que suportam os assistentes como o ChatGPT e similares. Como explicou Łukasz Kaiser [103], em um certo sentido a arquitetura *Transformer* é mais simples que as RNNs. Não há recorrências nem convoluções - *a Atenção é tudo o que é preciso*. Como veremos, a Atenção resulta de cálculos (multiplicações) com matrizes *realizados em paralelo*, em grande velocidade (alta eficiência computacional), e de uma operação softmax.

*As redes neurais Transformer são tão incríveis que parecem mágicas.*



### 4.3. A caixa mágica

A matriz de *tokens* utilizada no treinamento do GPT-3 tem  $T = 2048$  *tokens* de "largura". Esta é a sua **janela de contexto** (*context window*). O número de linhas é  $B$ , formando uma matriz  $B \times T$  podendo conter milhões de *tokens*.

O exemplo extraído de [104] mostra a frase de entrada contendo os *tokens* em verde "**Cite a primeira lei da robótica**" (*Input: Recite the first law of robotics*) sendo **completada** um token de cada vez pela frase composta pelos *tokens* em rosa "**Um robô não deve ferir um ser humano**" ("A robot may not injure a human being").

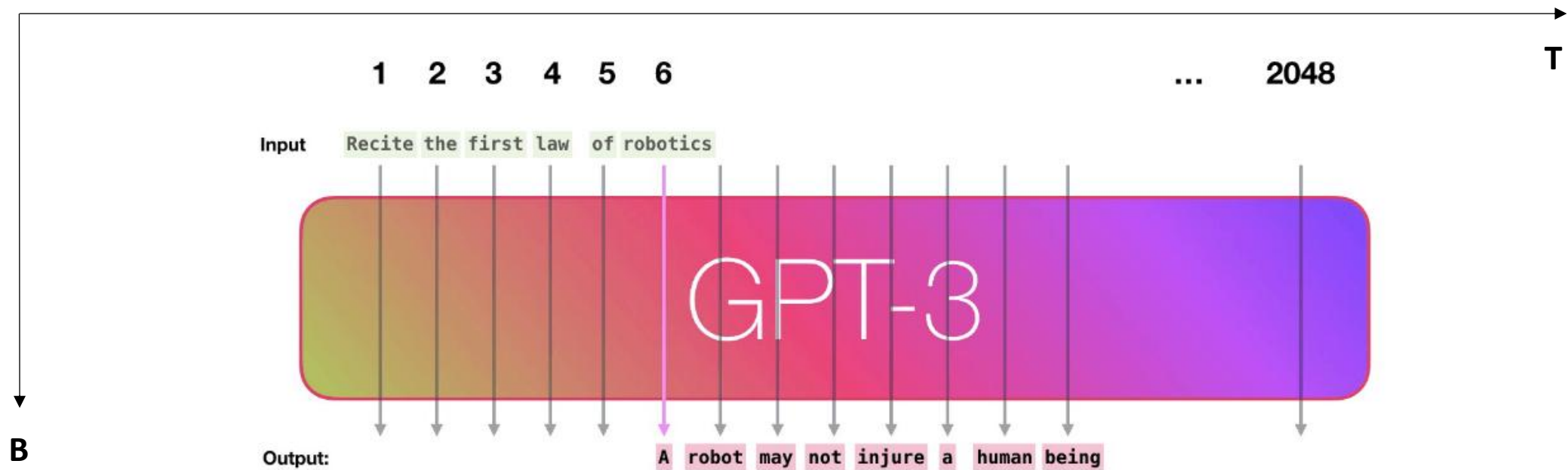
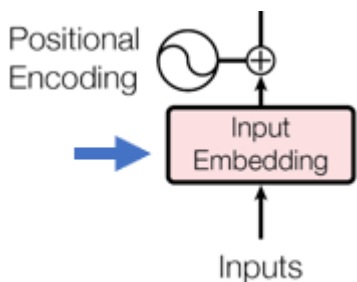


Figura 106 - Adaptada de [104] pelo autor

1 2 3 4 5 6

Recite the first law of robotics

Vetor de entrada com representação do *token* "robotics" + codificação posicional (#6)



1

O *token* "robotics" é **vetorizado**, ou seja, convertido em um **vetor de entrada** (amarelo) com números reais. Também ocorre uma codificação posicional (*Embedding posicional*) indicando a posição relativa de cada *token* (no caso, # 6).



Figura 107 - Adaptada de [104] pelo autor

2

O **vetor de entrada** (amarelo) representando o *token* "robotics" é passado para a rede neural *Transformer*, que no diagrama é mostrada como uma **Caixa Mágica**.



Figura 108 - Adaptada de [105] pelo autor

3

A rede *Transformer* faz a sua "mágica" e prediz o *token* com maior probabilidade para completar a frase. É gerado um **vetor de saída** (em azul) para representar este *token*.

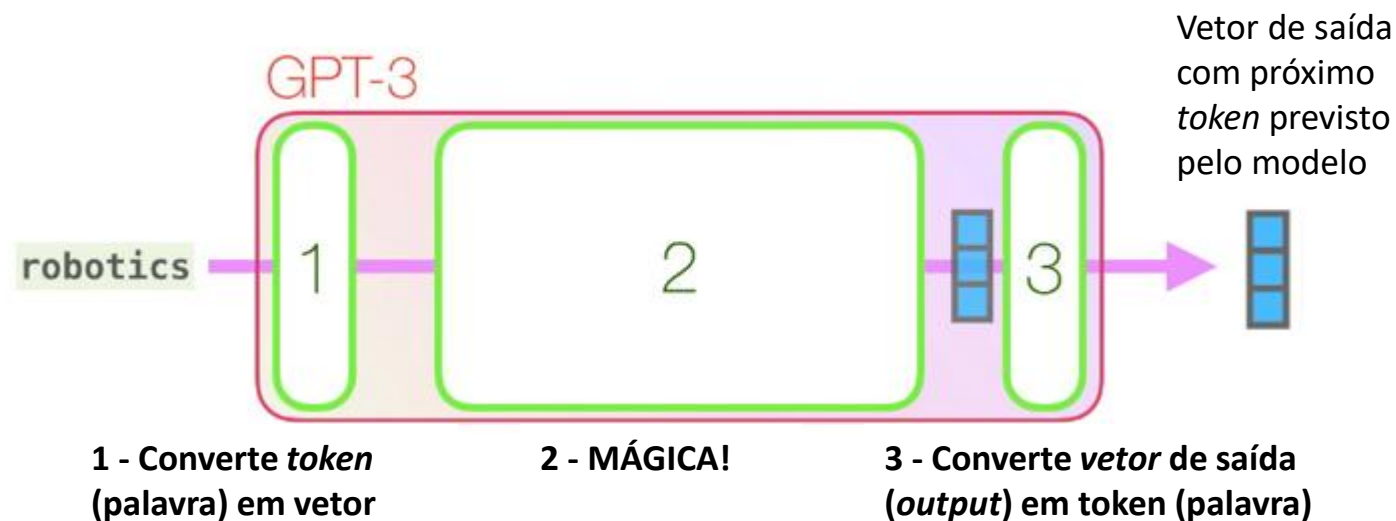


Figura 109 - Adaptada de [104] pelo autor

- 4 O **vetor de saída** é reconvertido em um *token* (no caso, a palavra mais provável é "A").



Figura 110 - Adaptada de [104] pelo autor

O processo segue, e novos *tokens* são gerados ("robot", "may", "not" etc.) até produzir a resposta completa.

Cite a primeira lei da robótica

Um robô não deve ferir um ser humano

Recite the first law of robotics → A robot may not injure a human being

Figura 111 - Adaptada de [104] pelo autor

*Como esta mágica acontece?*

Recite the first law of robotics

A Figura 112 mostra a arquitetura da rede neural Transformer.

A Transformer é baseada apenas em "Mecanismos de Atenção" (ver **Multi-Head Attention** no diagrama), sem a necessidade de recorrências e convoluções como nas redes neurais recorrentes, que operavam de forma sequencial.

Esta simplificação permitiu o paralelismo (processamento em paralelo), reduzindo o tempo de treinamento dos grandes modelos de linguagem como o GPT-3 e muitos outros.

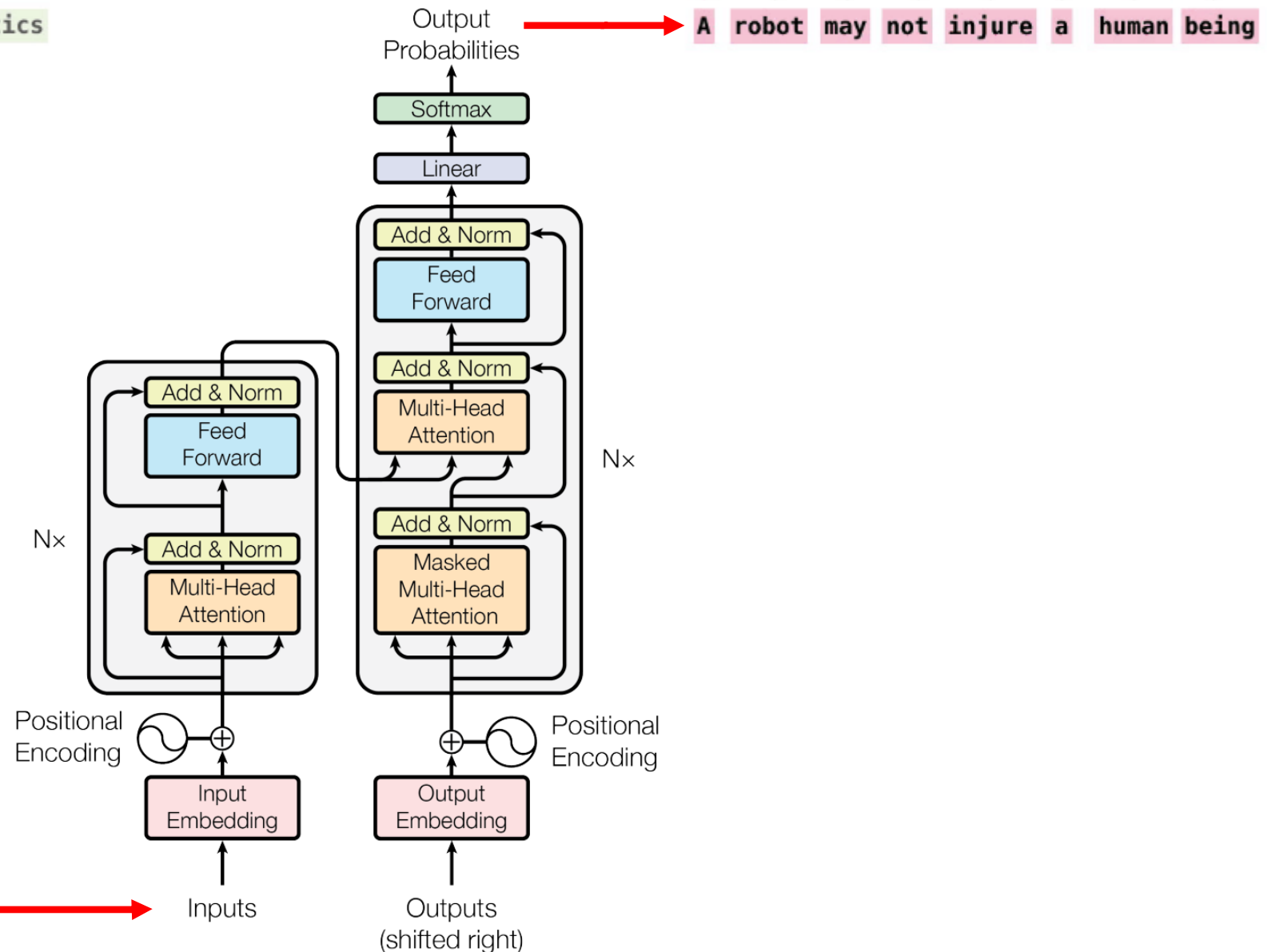
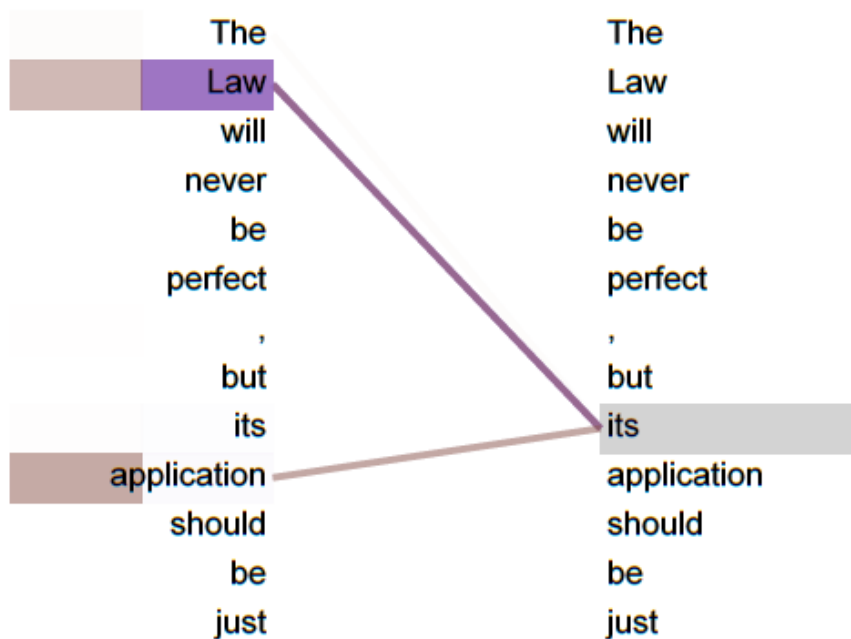


Figura 112 - Fonte: [80]

## 4.4. O mecanismo de Atenção

A relação semântica ou contextual entre palavras de uma sentença precisa ser identificada independentemente da ordem (que pode inclusive variar em diferentes idiomas) e da distância entre palavras. Por exemplo, em "the blue chair" (a cadeira azul) o adjetivo ("blue") aparece antes do nome ("chair"), mas na tradução ao Português o adjetivo "azul" vem depois do nome. Assim, para traduzir a *próxima* palavra (*token*) em uma sentença de forma adequada, é geralmente útil também levar em conta ("prestar atenção") outras palavras em *posições diferentes* na sentença.



De forma simplificada, o **mecanismo de Atenção** permite que a rede neural "preste atenção" em toda a sequência de *tokens* passada como entrada, mas de modo diferente para cada palavra. Isso ajuda a produzir de forma mais apropriada a *próxima saída* (próximo *token*), pois a Atenção permite levar em conta a proximidade (ou relevância) dos diferentes termos da sentença *independentemente da sua distância (posição)*.

Figura 113 - Na sentença "A **Lei (Law)** nunca será perfeita, mas sua (its) aplicação deve ser justa" o **mecanismo de Atenção** é bem preciso em destacar que a palavra "sua" ("its") deve "prestar muita atenção" na palavra "Lei", *que aparece bem antes na sentença*, e também alguma atenção na palavra "aplicação" ("application") que vem em seguida.

Figura 113 - Fonte: Adaptada de [80] pelo autor

Suponha que desejamos traduzir a sentença "O animal não atravessou a rua porque ele estava muito cansado". A que a palavra "ele" ("it") se refere nesta sentença? Quem está muito cansado? Para nós humanos, é fácil inferir que o pronome "ele" se refere ao animal, mas não é tão simples para uma rede neural fazer esta inferência.

Quando uma sentença atravessa a rede *Transformer*, os **mecanismos de Atenção** calculam a similaridades (semântica) entre as diferentes palavras e geram **pesos de Atenção**.

Figura 114 - Para cada palavra em consulta (no caso da figura, `it_` ou `ele_`) é gerado um número que indica o nível de atenção que a palavra consultada deve dar a si própria, e também a todas as demais palavras da sentença durante a tarefa em execução (por exemplo, uma tradução).

Isso permite que a palavra em consulta ("ele") preste *muita* atenção na palavra "animal", e atente um pouco menos nas demais palavras da sentença.

Na figura, as **associações com maior peso** são mostradas em **cores mais escuras**.

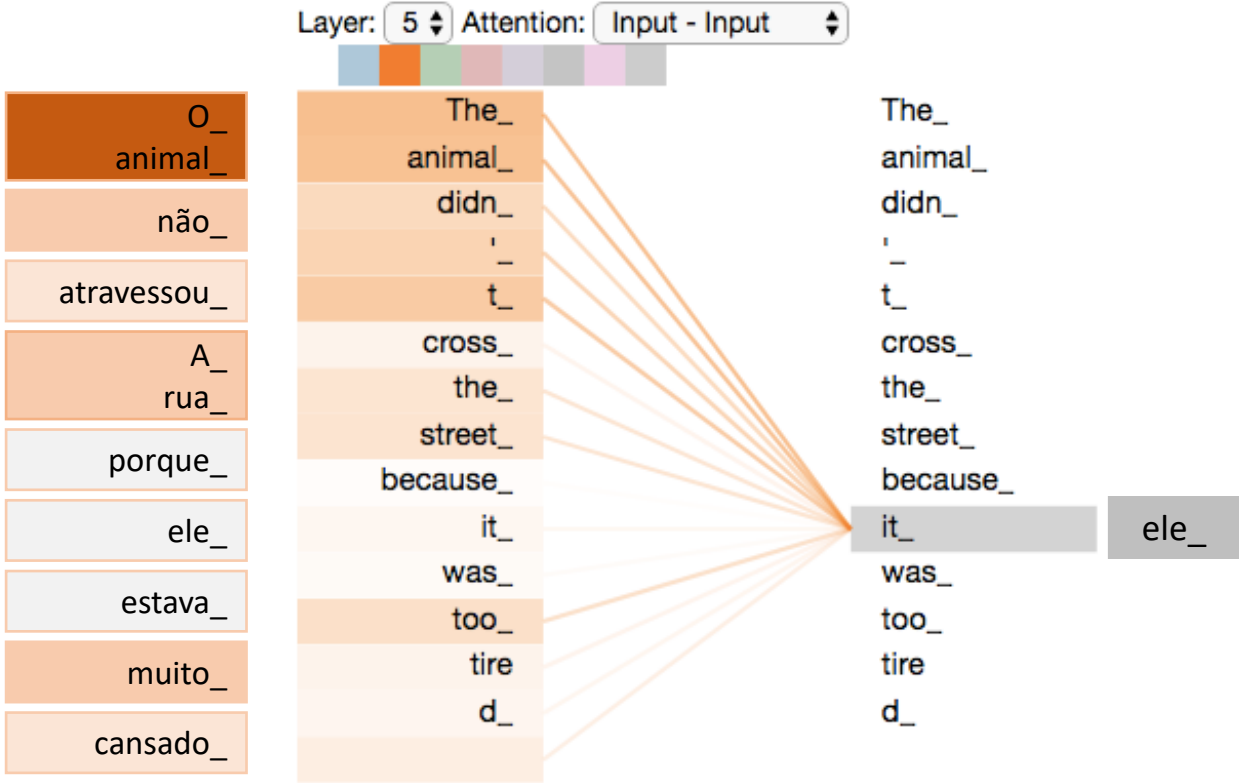


Figura 114 - Fonte: Adaptada de [80] pelo autor

## 4.5. A arquitetura *Transformer*

A arquitetura *Transformer* foi proposta no artigo "*Attention is all you need*" [80] em 2017 por Ashish Vaswani e outros pesquisadores da Google. Foi originalmente pensada para ajudar em tarefas de tradução de idiomas, mas tem sido utilizada em várias outras tarefas de processamento de linguagem natural, e mesmo em outras áreas do *Deep Learning* como a Visão Computacional. O assunto é bem complexo (estamos falando sobre a arquitetura estado da arte para treinar modelos utilizados em inteligência artificial), e os interessados em uma abordagem mais técnica podem recorrer a [80, 104, 105, 106, 107 e 108 (abordagem matemática)]. Por hora, vamos passar a ideia básica, seguindo passo a passo o conteúdo compartilhado por Jay Alammam em seu artigo "*The Illustrated Transformer*" [105], onde o autor utiliza animações para explicar conceitos técnicos de forma simples, utilizando uma tarefa de tradução como exemplo.

Primeiro, vamos analisar o modelo como uma Caixa Preta única.

Em uma tarefa de tradução de idiomas, o modelo recebe a sentença em Francês **Je suis étudiant** (eu sou estudante) como entrada (*input*), e entrega a tradução em Inglês (**I am a student**) como saída (*output*).



Figura 115 - Fonte: Adaptado de [105] pelo autor



Abrindo um pouquinho a Caixa Preta onde a mágica acontece, vemos um bloco de codificação (ENCODERS), um outro bloco de decodificação (DECODERS), e conexões entre eles (Figura 116).

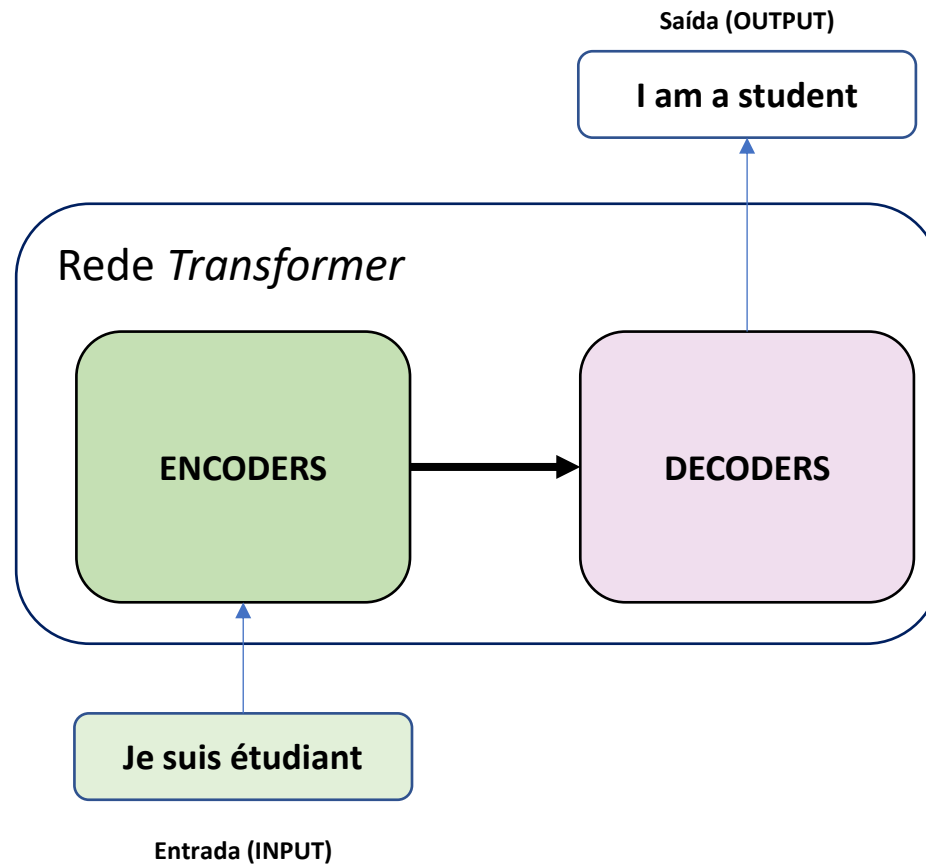


Figura 116 - Fonte: Adaptada de [105] pelo autor

No artigo original por Vaswani e outros [80], o bloco de codificadores possui uma pilha com seis ENCODERS (cada um é uma rede neural). Da mesma forma, o bloco de decodificação possui uma pilha com seis DECODERS (cada um é uma rede neural). Não é necessário que sejam exatamente seis, outras configurações são possíveis. Mas vamos seguir a especificação original.

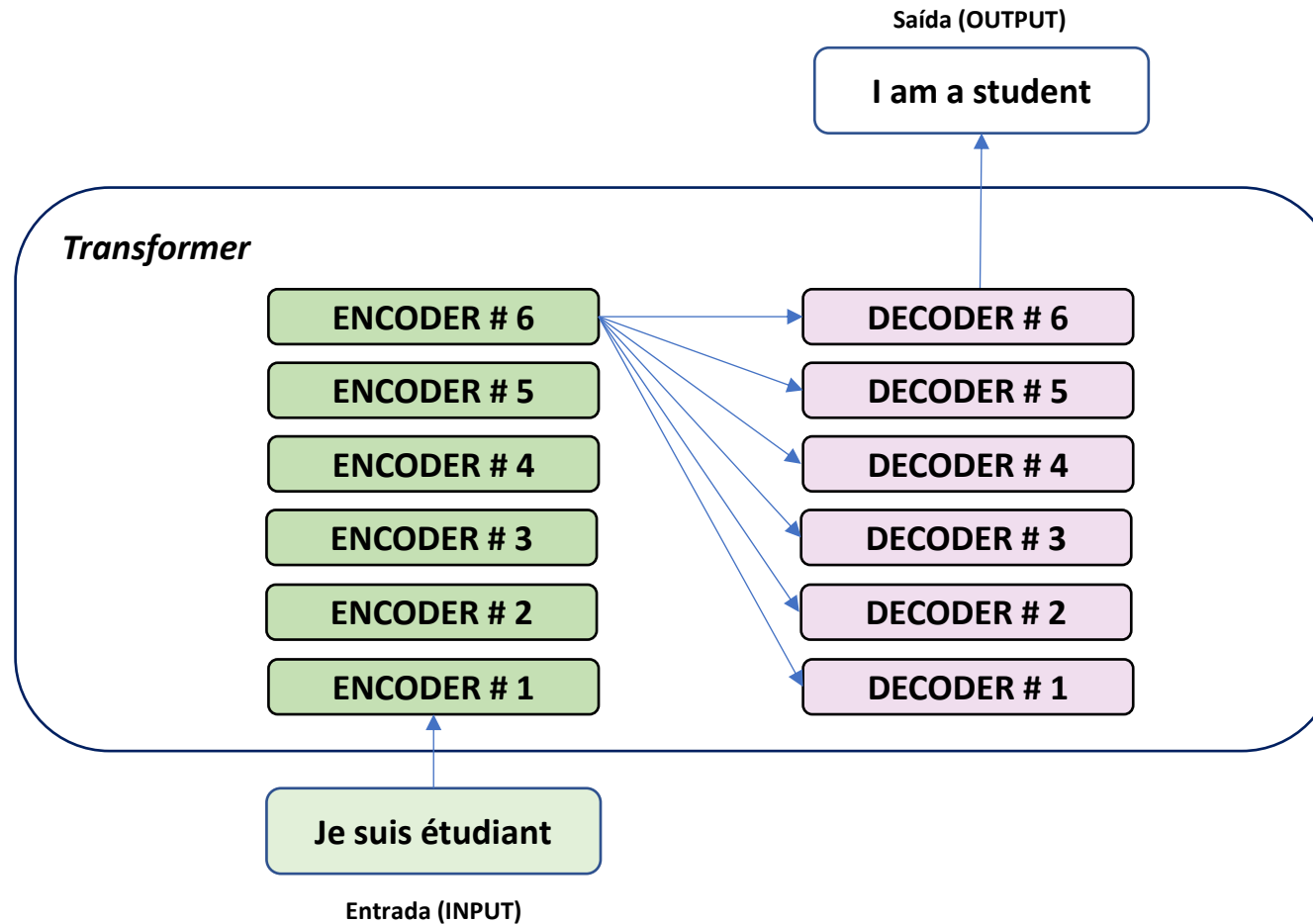


Figura 117 - Fonte: Adaptada de [105] pelo autor

### 4.5.1. ENCODERS

Cada ENCODER é dividido em duas subcamadas. As entradas (INPUTS) passam primeiro pela subcamada de **Auto-Atenção** (*Multi-Head Attention*), que ajuda o ENCODER a analisar todas as palavras (*tokens*) da sequência de entrada, enquanto codifica cada palavra. As saídas da subcamada de Auto-atenção são enviadas para a outra subcamada, uma Rede Neural *Feed-Forward* (FFNN). O mesmo tipo de rede FFNN é utilizado em cada um dos seis ENCODERS na pilha.

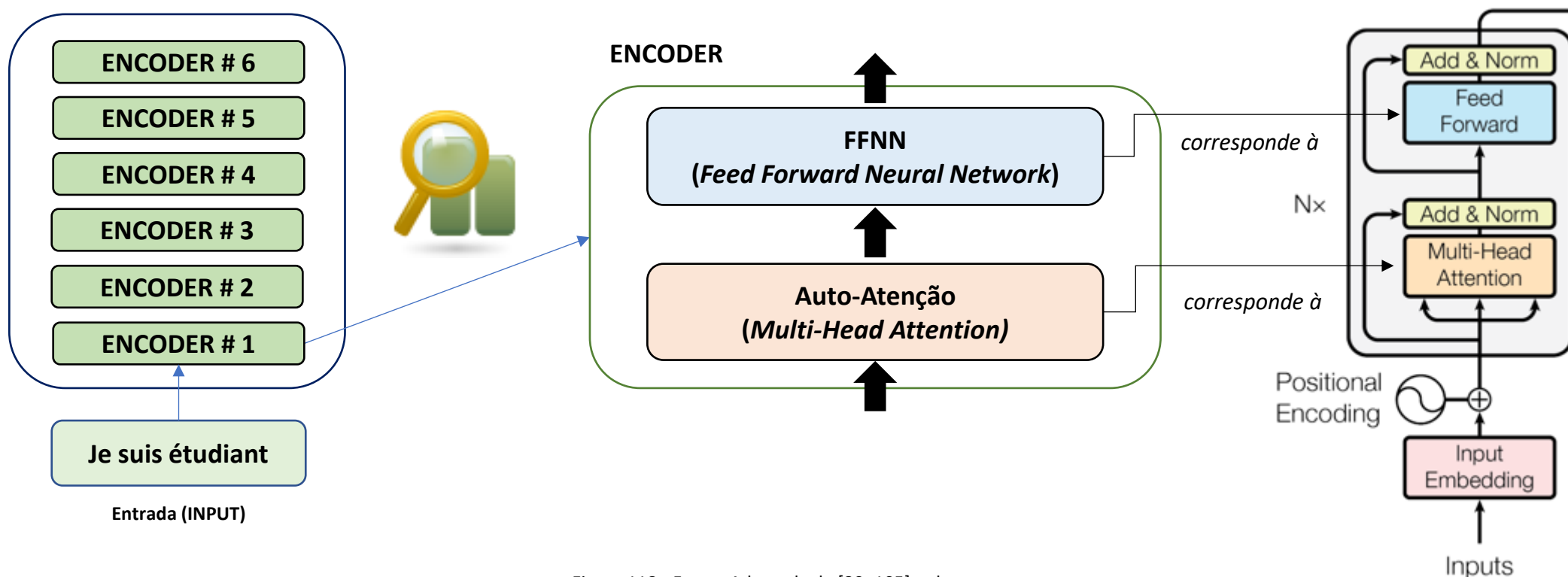


Figura 118 - Fonte: Adaptada de [80, 105] pelo autor

O primeiro passo em uma tarefa de tradução por rede *Transformer* (bem como em várias outras tarefas de processamento de linguagem natural) é a **vetorização**, ou transformar cada palavra (*token*) em um vetor de números reais, utilizando um **algoritmo de *Embedding***.

No exemplo da Figura 119 extraída diretamente de [105], cada palavra da sentença em Francês que se deseja traduzir (Je suis étudiant = eu sou estudante) é "embutida" (*Embedded*) em um **vetor  $x$  de *Embedding*** (caixinhas verdes).

Este processo de vetorização ou *Embedding* ocorre apenas uma vez, no ENCODER de nível mais baixo na pilha.

Concluído o *Embedding* das palavras da sequência de entrada (inclusive a *parte posicional gerada por sinais temporais* - mais sobre isso logo a seguir), cada **vetor  $x_j$**  representando cada palavra  $j$  da sequência entra no primeiro ENCODER do bloco, através da subcamada de **Auto-Atenção (*Self-Attention*)**.

Como veremos, desta subcamada saem **vetores de atenção  $z_j$** .

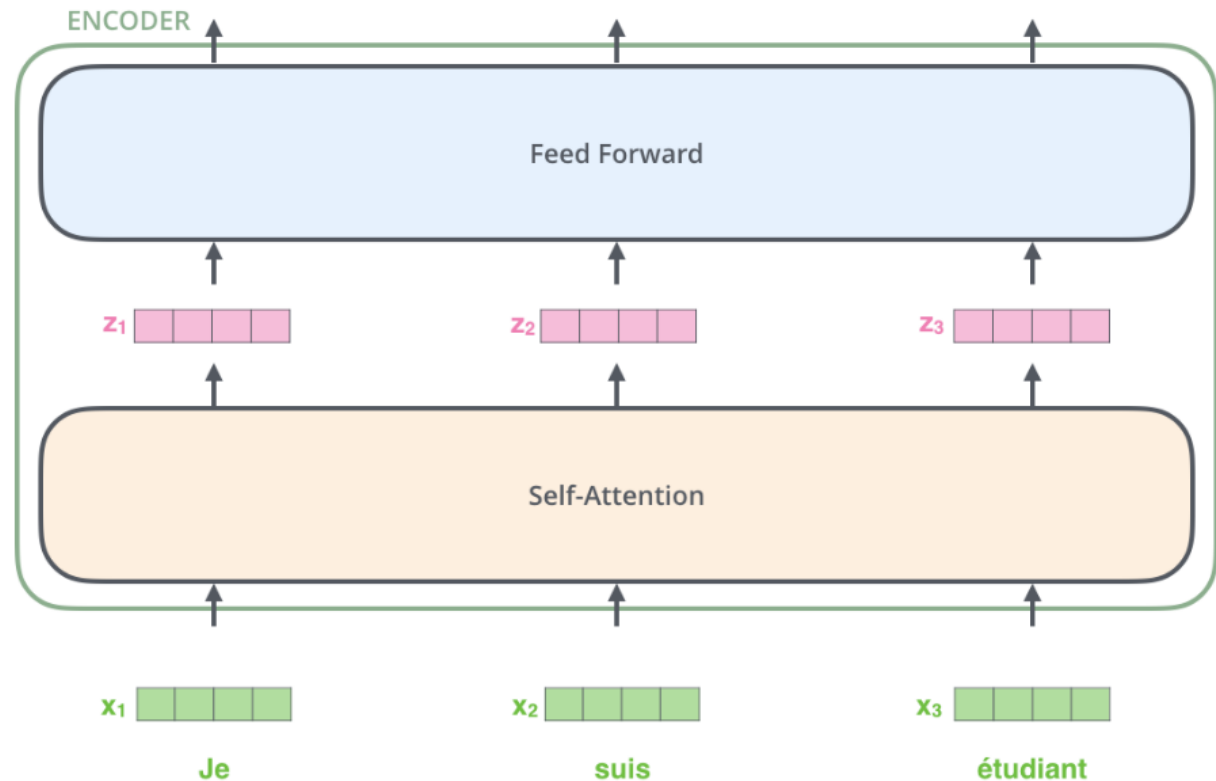


Figura 119 - Fonte: [105]

#### 4.5.1.1. Codificação posicional dos vetores de Embeddings

No processo de codificação posicional, **vetores  $t$**  que seguem um padrão bem definido de geração temporal (sinais gerados por uma função senoidal por exemplo) são adicionados aos **vetores  $x$  de Embeddings**, gerando um **novo vetor  $x$**  que podemos chamar de "*Embedding com sinal temporal*" (BOX 7). Além do "*Embedding*", este novo vetor permite que a rede *Transformer* controlar a ordem dos diferentes *tokens* na sentença, e portanto é ele que é passado como entrada para o primeiro ENCODER (#1) da pilha. Porém, em favor da simplicidade, em vez de falar de "*vetor  $x$  com sinal temporal*" ou "*vetor  $x$  de Embedding posicional*" vamos nos referir apenas ao "**vetor  $x$  de Embedding**" daqui por diante.

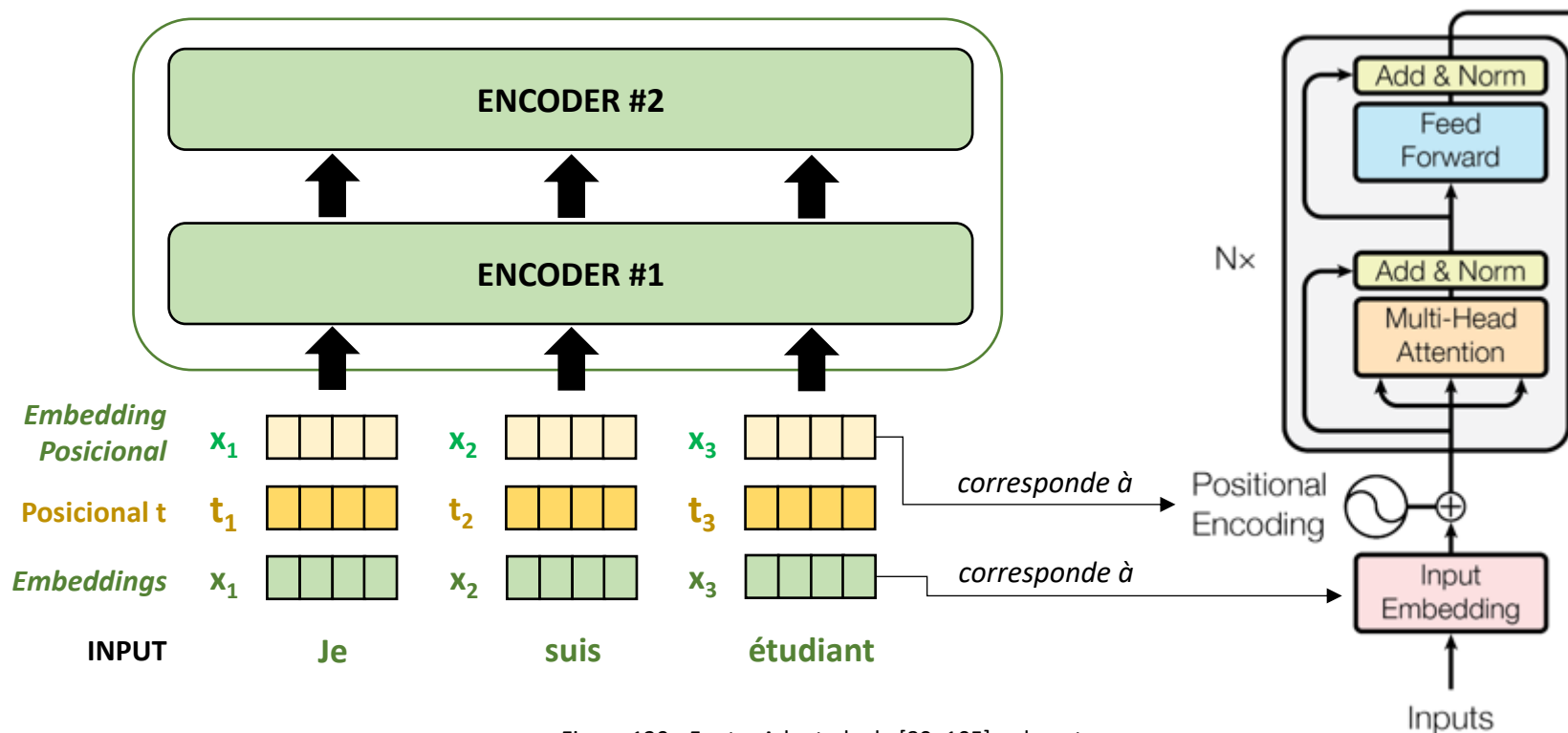


Figura 120 - Fonte: Adaptada de [80, 105] pelo autor

## BOX 7. Vetores posicionais e vetores de *Embeddings*

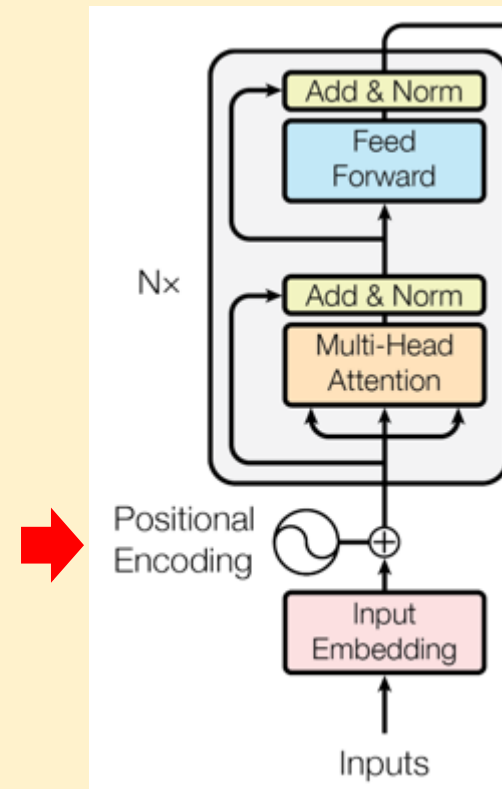
Na arquitetura *Transformer*, os **vetores posicionais**  $\mathbf{t}$  são adicionados aos **vetores  $\mathbf{x}$  de *Embedding*** para fornecer informações sobre a posição relativa das palavras na sequência de entrada. Isso é necessário porque o *Transformer* não possui uma estrutura recorrente que naturalmente codifica a ordem sequencial das palavras (como era o caso nas RNNs, ou *Redes Neurais Recorrentes*). A função que gera os vetores posicionais é chamada de **função de posicionamento** ou **codificador posicional** (*Positional Encoding*).

A forma mais comum de gerar os **vetores posicionais**  $\mathbf{t}$  é por meio de funções trigonométricas, como senos e cossenos, que calculam um vetor posicional para cada posição na sequência de entrada. Em seguida, esse vetor é somado ao **vetor  $\mathbf{x}$  de *Embedding*** correspondente.

$$\text{positional\_encoding}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

$$\text{positional\_encoding}(\text{pos}, 2i + 1) = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

Onde  $\text{pos}$  é a posição na sequência,  $i$  é a dimensão do **vetor posicional** e  $d$  é a dimensão do **vetor de *Embedding***. Este mecanismo *fornece uma representação única para cada posição na sequência, permitindo que o modelo leve em consideração a ordem das palavras*. A soma dos **vetores  $\mathbf{t}$  posicionais** aos **vetores  $\mathbf{x}$  de *Embedding*** é realizada antes de serem alimentados no primeiro ENCODER do *Transformer*.



Jay Alammar chama atenção para uma propriedade chave da arquitetura *Transformer*: cada palavra (*token*) em cada posição da sentença segue um caminho próprio ao atravessar a pilha de ENCODERS.

Existem dependências entre estes caminhos na subcamada de Auto-Atenção, já que existem relações (de significado) entre as diferentes palavras. Já na subcamada com a Rede Neural *Feed-forward* estas dependências entre as palavras *não são levadas em conta*, e desta forma os vários caminhos de processamento de cada vetor (cada palavra) podem ser **executados em paralelo** nesta subcamada - o que acelera bastante o processo de treinamento dos modelos.

Figura 122 - O ENCODER #1 recebe uma lista de **vetores  $x_j$  de *Embeddings*** (já posicionais!) como entrada, e processa esta lista passando os vetores (um de cada vez) para a subcamada de Auto-Atenção (*Self-attention*), dando origem aos **vetores de atenção  $z_j$** .

Em seguida, ainda no ENCODER #1, o **vetor  $z_j$**  segue para a subcamada com a rede neural *Feed-forward*, cuja saída é o **vetor de representação  $r_j$**  que é enviado para o próximo ENCODER #2.

O que são estes vetores, e para que servem? É o que veremos a seguir.

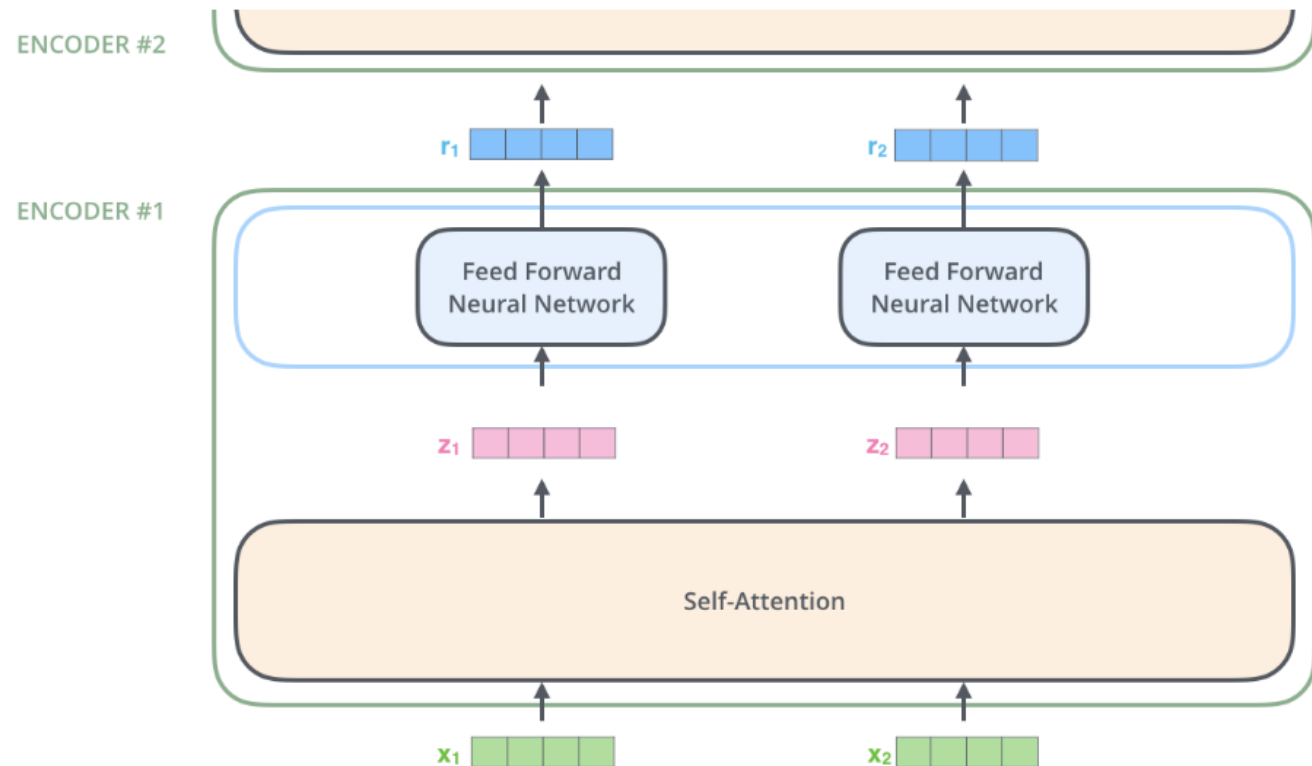


Figura 122 - Fonte: [105]

#### 4.5.1.2. Vetorização

Vejam os como o número (peso) que indica a Atenção é calculado. Cada **vetor de Embedding**  $x_i$  representando cada palavra (*token*) na posição  $i$  da sequência é associado a três outros vetores ( $q_i$ ,  $k_i$ ,  $v_i$ ):

**Vetor  $x$**  —————> Associado a **Vetor  $q$**  - Consulta (*Query*), **Vetor  $k$**  - Chave (*Key*) e **Vetor  $v$**  - Valor (*Value*)

Os vetores ( $q_i$ ,  $k_i$ ,  $v_i$ ) são criados multiplicando-se o **vetor  $x_i$  de Embedding** de cada palavra na posição  $i$  por três matrizes de pesos ( $W^Q$ ,  $W^K$ ,  $W^V$ ) que são geradas durante o processo de treinamento do modelo. Ou seja, generalizando para qualquer posição  $i$  temos  $q_i = x_i \cdot W^Q$ ,  $k_i = x_i \cdot W^K$  e  $v_i = x_i \cdot W^V$ . Por exemplo, o produto do **vetor  $x_1$**  (*Embedding* da palavra na posição 1) pela matriz de pesos  $W^Q$  gera o **vetor de Consulta  $q_1$**  associado com a primeira palavra (Figura 123).

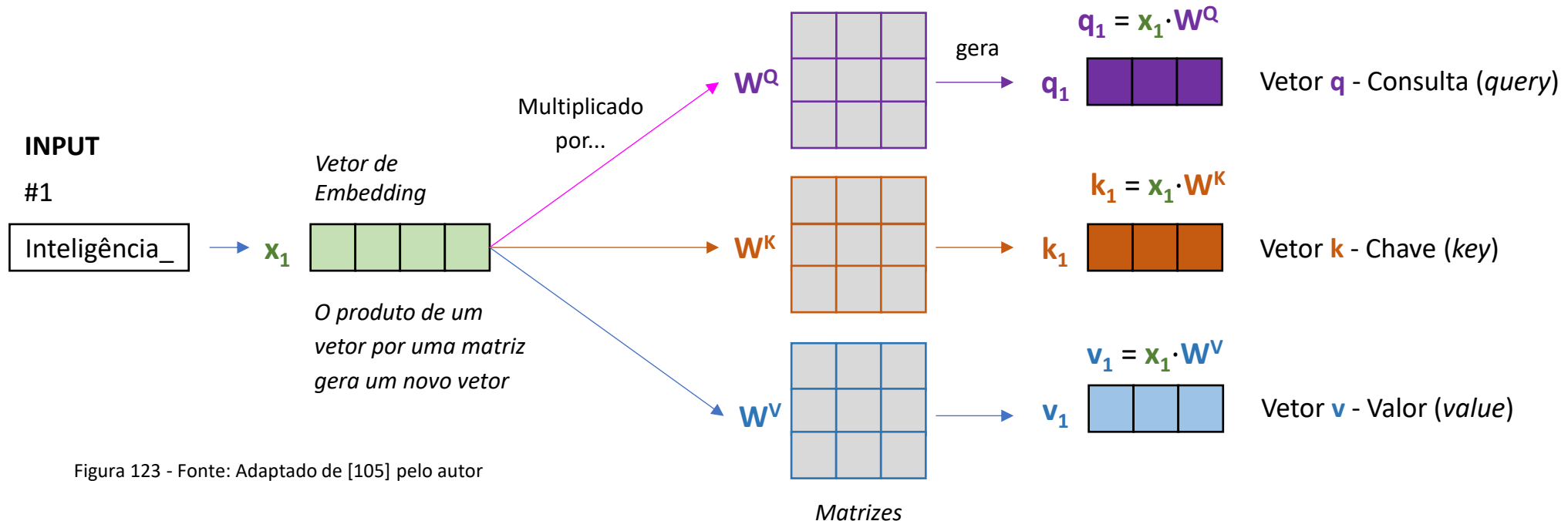


Figura 123 - Fonte: Adaptado de [105] pelo autor



Imagine que temos uma sentença com apenas duas palavras, "Inteligência Artificial", para simplificar. Para a primeira palavra ("Inteligência"), o produto do vetor  $x_1$  pela matriz de pesos  $W^Q$  gera o vetor  $q_1$ , o produto do vetor  $x_1$  pela matriz  $W^K$  gera o vetor  $k_1$  e o produto do vetor  $x_1$  pela matriz  $W^V$  gera o vetor  $v_1$ . Da mesma forma, para a segunda palavra da sentença ("Artificial"), o produto do vetor  $x_2$  pela matriz de pesos  $W^Q$  gera o vetor consulta  $q_2$  etc. A Figura 124 mostra os vetores  $(q_1, k_1, v_1)$  e  $(q_2, k_2, v_2)$  gerados a partir dos vetores  $x_1$  e  $x_2$  de *Embedding* das duas palavras da sentença.

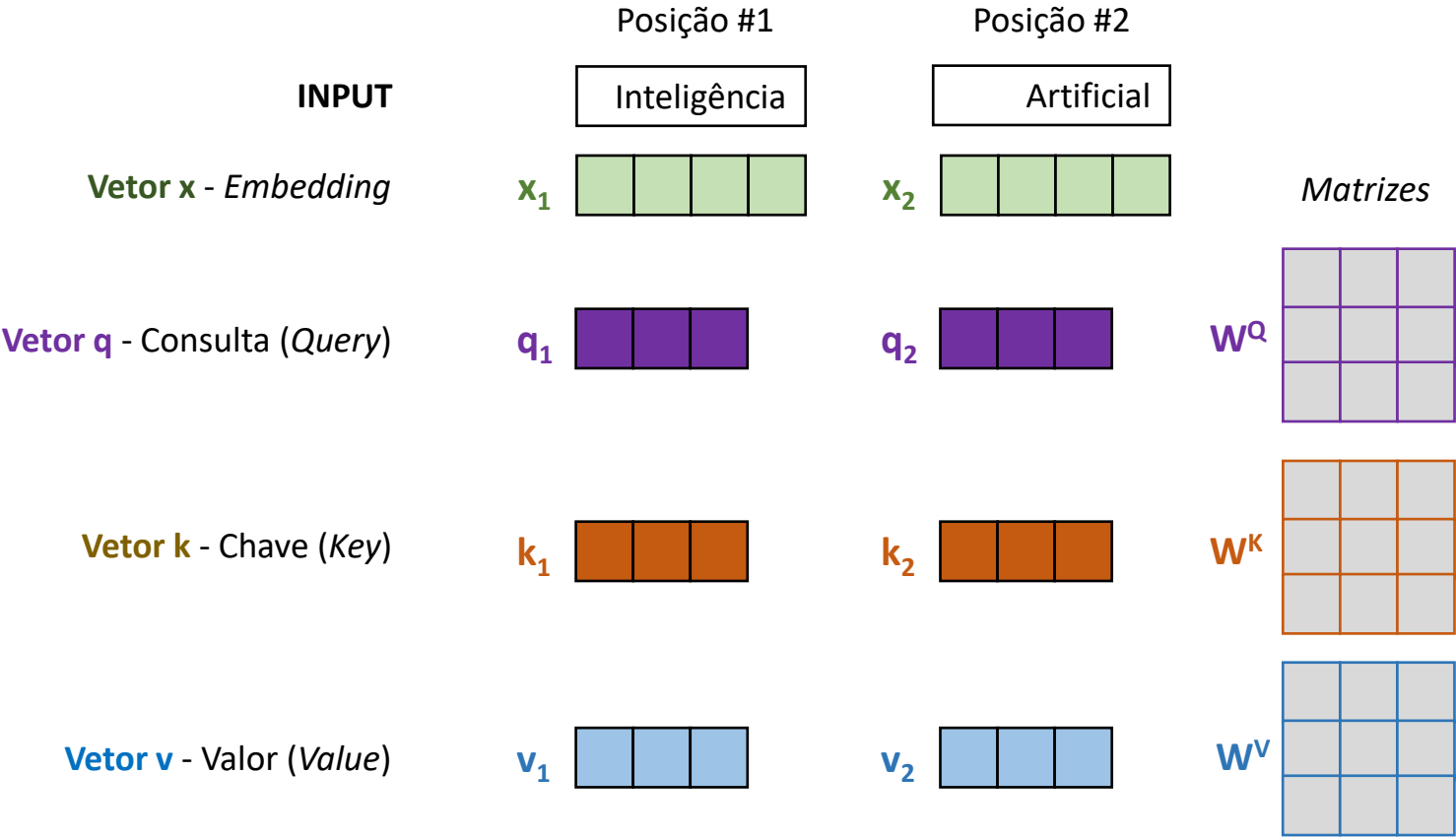


Figura 124 - Fonte: Adaptada de [105] pelo autor

Para que se servem todos estes vetores?

Para o cálculo da Atenção. Há várias palavras em diferentes posições em uma sentença. Qual é a importância de cada uma para as demais? A Atenção indica o quão importante é a informação da palavra na posição  $j$  da sequência para a palavra na posição  $i$ .

Aqui é importante entender os conceitos de **palavra de consulta** e **palavras de chave**.

- Em uma sequência com várias palavras, a **palavra de consulta** é a palavra que está "perguntando" o quanto ela deve atentar, ou "prestar atenção" em cada palavras da sequência (*inclusive ela própria*). Portanto, a Atenção é modelada a partir da perspectiva da **palavra de consulta**.
- Escolhida uma palavra de consulta na sequência, todas as demais são **palavras de chave** (representadas por seus **vetores de chave**,  $k$ ). Estas são as palavras que estão sendo "consideradas" para determinar o quão relevante elas são para a **palavra de consulta** na posição atual.

A sequência abaixo tem seis palavras. "anéis" ocupa a Posição #2, "Saturno" a Posição #4, e "belos" a Posição #6. Podemos escolher por exemplo a palavra na Posição 2 ("anéis") como **palavra de consulta** destacada no *box* preto.



Figura 125

Neste caso, podemos calcular *quanto de atenção a palavra "anéis" deve prestar nela própria, e também nas outras palavras*, representadas por seus respectivos **vetores de chave**. Por exemplo, quanto de Atenção a palavra "anéis" deve prestar na palavra "Saturno", ou na palavra "belos"?

**Palavra de consulta:** "anéis" (posição #2 na sequência).

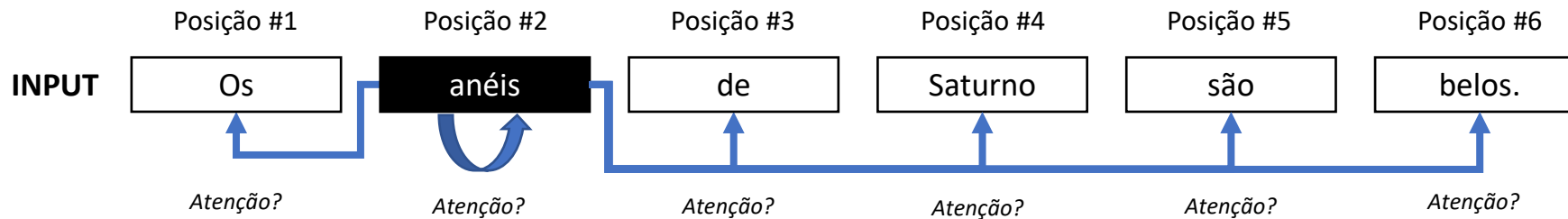


Figura 126

Podemos calcular a Atenção  $A_{tt}(i, j)$  que "anéis" (em consulta, na posição  $i$ ) deve dar a si própria e cada uma das outras palavras restantes nas posições  $j$  da sequência ("Os", "de", "Saturno", "são", "belos"). *Este processo é realizado para todas as palavras, ou seja, todas são colocadas "em consulta", e é calculado o Score de Atenção que cada palavra deve prestar em todas as palavras da sentença.*

É aqui que entram os vetores. A pontuação de Score da Atenção  $Att(i, j)$  que uma **palavra em consulta** na posição  $i$  da sentença deve dedicar a alguma outra **palavra de chave** na posição  $j$  da sentença é calculada pelo produto escalar do **vetor de consulta**  $q_i$  da **palavra de consulta** na posição  $i$  pelo **vetor chave**  $k_j$  que representa a **palavra de chave** na posição  $j$ .

$$Score_{Att}(i, j) = q_i \circ k_j$$

Assim, por exemplo, o Score de Atenção da **palavra em consulta** na posição  $i = 2$  ("anéis") em relação à **palavra de chave** na posição  $j = 4$  ("Saturno") será um número (o resultado do produto escalar entre dois vetores é um escalar, ou seja, um número - veja o BOX 8), por exemplo, 112.

$$Score_{Att}(2, 4) = q_2 \circ k_4 = 112$$

## BOX 8. Produto escalar de dois vetores

O produto escalar [109], também conhecido como produto interno entre dois vetores  $\mathbf{u}=(u_1,u_2,\dots,u_n)$  e  $\mathbf{v}=(v_1,v_2,\dots,v_n)$  de dimensão  $n$  (ambos os vetores têm  $n$  elementos) é calculado pela soma dos produtos de seus componentes correspondentes. A fórmula para o produto escalar entre  $\mathbf{u}$  e  $\mathbf{v}$  é a seguinte:

$$\mathbf{u} \cdot \mathbf{v} = u_1 \cdot v_1 + u_2 \cdot v_2 + \dots + u_n \cdot v_n$$

Ou em uma notação um pouco mais compacta

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^n u_i \cdot v_i$$

O resultado do produto escalar é um número real. Além da definição algébrica dada acima, há também uma interpretação geométrica: o produto escalar entre os vetores  $\mathbf{A}$  e  $\mathbf{B}$  é dado  $|\mathbf{A}| \cdot |\mathbf{B}| \cos \theta$  onde  $\theta$  é o ângulo entre os vetores, e  $|\mathbf{A}|$  e  $|\mathbf{B}|$  são os módulos ou comprimentos dos vetores  $\mathbf{A}$  e  $\mathbf{B}$ . Se o resultado é zero, isso significa que os vetores são ortogonais (perpendiculares), pois  $\cos 90^\circ$  é zero. Se o resultado é positivo, os vetores estão na mesma direção, como no exemplo da Figura 127, e se é negativo, os vetores estão em direções opostas. O produto escalar é uma operação fundamental em álgebra linear e é frequentemente usado em várias disciplinas, incluindo o *Machine Learning*.

$|\mathbf{A}| \cos(\theta)$  é a projeção escalar do vetor  $\mathbf{A}$  sobre o vetor  $\mathbf{B}$ .

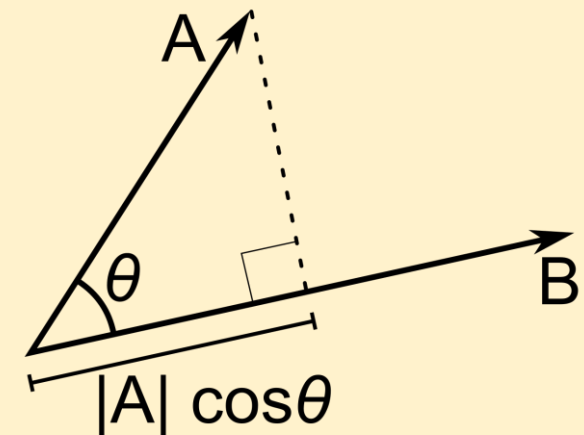


Figura 127 - Fonte: [109]

Naturalmente, nem todas as palavras da sequência devem receber a mesma Atenção, seja em uma tarefa de tradução ou outra tarefa qualquer.

Em primeiro lugar, para qualquer tarefa que vá ser realizada com a **palavra de consulta**, ela própria deve ser levada em conta obviamente, ou seja, *as palavras em consulta também prestam atenção em si mesmas*. De fato, é esperado que o valor associado à **Atenção própria** seja mais elevado do que os valores associados às outras palavras na sequência. Afinal de contas, cada palavra deve considerar sua própria informação como mais relevante para ela mesma, além de poder considerar também a informação de *outras* palavras da sentença como mais ou menos relevantes para ela.

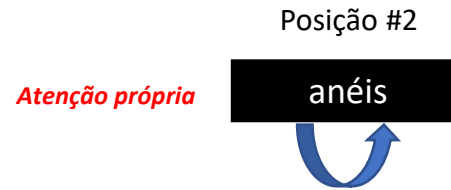


Figura 128

Com relação às outras palavras, considerando o exemplo em questão, quando "anéis" está sendo consultada espera-se intuitivamente que "Saturno" receba um valor de Atenção mais elevado do que as outras palavras ("Os", "de", "são" e "belos"), pois é comum na linguagem encontrar referências para "anéis de Saturno", logo há forte similaridade (proximidade) entre "anéis" e "Saturno". Também existe alguma proximidade entre "anéis" e "belos" (anéis em geral são objetos belos), e portanto "anéis" também deve prestar alguma Atenção em "belos", mas já não há tanta similaridade entre as palavras "anéis" e "são" por exemplo.

Com isso em mente, podemos prosseguir, pois é um pouquinho mais complicado. O valor da Atenção *ainda não é* este *Score* calculado pelo produto escalar de dois vetores - algumas outras operações matemáticas precisam ser feitas.

4.5.1.3. O cálculo da Atenção

Passo 1 - Cálculo do Score da Atenção

O primeiro passo no cálculo da Atenção é justamente o que acabamos de discutir - o cálculo do *Score*. Como vimos, a pontuação do *Score* da Atenção que a palavra na posição *i* da sentença (**palavra de consulta**) deve prestar em qualquer palavra na posição *j* é calculada pelo produto escalar

$$Score_{Att}(i,j) = q_i \circ k_j$$

Onde  $q_i$  é o **vetor de consulta** para a **palavra de consulta** na posição *i*, e  $k_j$  é o **vetor de chave** que representa cada **palavra de chave** na sequência na posição *j*. A Figura 129 destaca apenas três palavras da sentença anterior para simplificar. A **palavra de consulta** é "anéis". São mostrados alguns valores para o *Score* de Atenção. Por exemplo, o valor do *Score* da Atenção que "anéis" deve prestar em si própria é 112, em "Saturno" é 96, e em "belos" é 24.

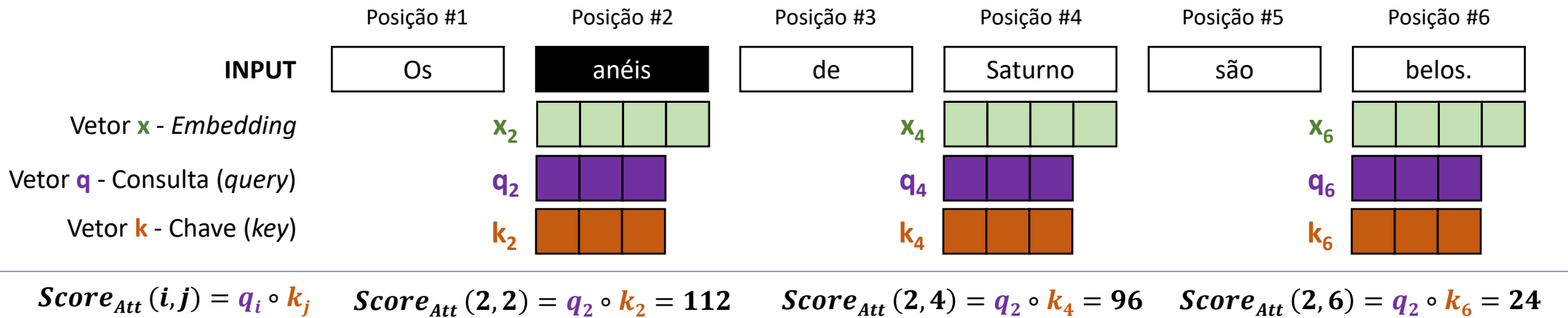


Figura 129 - Fonte: Adaptada de [105] pelo autor

## ***Passo 2 - Dimensionalidade dos vetores de consulta e de chave e normalização***

Os vetores de números reais ( $q$ ,  $k$ ,  $v$ ) possuem várias dimensões (a dimensão de um vetor é a quantidade de elementos que possui). Quando os vetores de chave  $k$  e de consulta  $q$  têm a mesma dimensão (que é um número inteiro), ela é indicada pela notação  $d_k$ .

A quantidade de dimensões dos vetores está relacionada à capacidade do modelo de aprender representações complexas. Dimensões mais altas podem permitir ao modelo capturar padrões mais sutis nos dados, mas também podem aumentar o custo computacional do modelo.

Em termos práticos, a dimensão  $d_k$  é um hiperparâmetro que pode ser ajustado durante o projeto do modelo. Em implementações comuns do *Transformer*, valores como 64, 128 ou 256 são frequentemente usados para  $d_k$ , dependendo da tarefa que se pretende realizar, e dos recursos computacionais disponíveis. Assim, por exemplo, se os vetores tiverem 64 dimensões, teremos  $d_k = 64$ .

A importância disso é que no cálculo da Atenção, o resultado do produto escalar entre os vetores  $q_i$  e  $k_j$  obtido no Passo 1 é dividido pela raiz quadrada de  $d_k$  ou  $\sqrt{d_k}$  para evitar que os valores fiquem muito grandes. Isso faz com que o treinamento do modelo seja mais estável.

Portanto, a pontuação normalizada para os *Scores* será dada por

$$Score' A_{ij} = \frac{q_i \circ k_j}{\sqrt{d_k}}$$

Temos agora então os *Scores* já normalizados pela dimensão  $d_k$ . Com  $d_k=64$ , temos  $\sqrt{d_k} = 8$ .

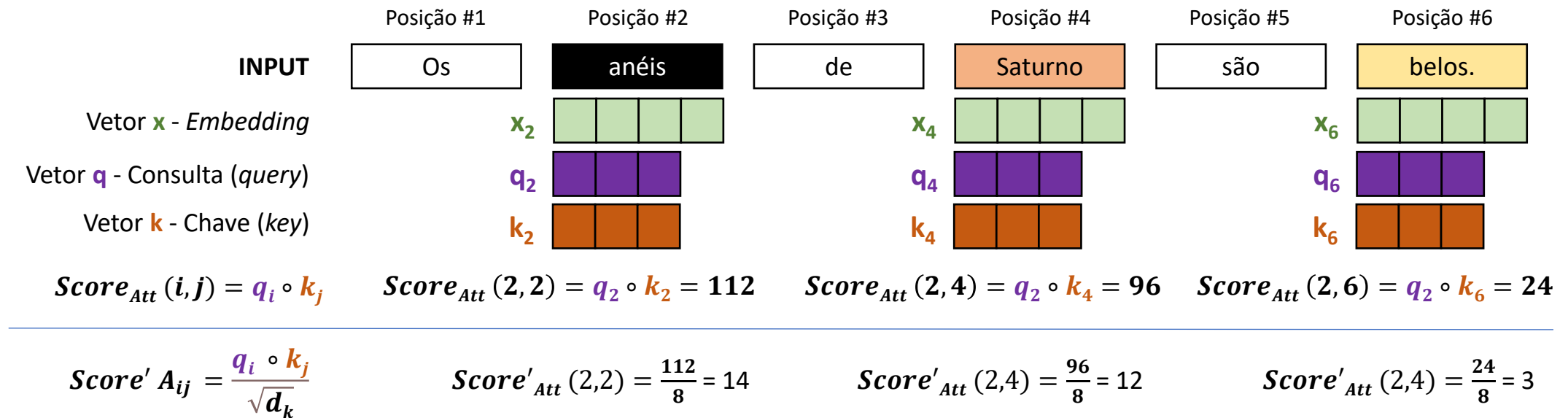


Figura 130 - Fonte: Adaptada de [105] pelo autor

Figura 130 - Com a palavra "anéis" em consulta, a rede *Transformer* atribui um *Score* de Atenção (já devidamente normalizado) mais alto para si própria (valor = 14). A autoatenção permite que a rede dê mais peso à informação localizada na própria palavra, capturando assim a dependência contextual da palavra em relação a si mesma. O segundo maior *Score* de Atenção foi atribuído para "Saturno" (com valor 12), indicando que esta palavra também deve receber Atenção, pois é relevante para a palavra de consulta, como era esperado. Finalmente, o *Score* atribuído para a palavra "belos" foi o menor (valor = 3), sugerindo um nível de Atenção menor (menos importante). As outras palavras da sequência ("Os", "de", "são") nas posições #1, #3 e #5 também recebem *Scores* de Atenção, mas em favor da simplicidade podemos imaginar que são tão baixos que podem ser ignorados.



**Passo 3 - A função softmax e o "peso" da Atenção**

A função softmax [73] é utilizada para normalizar os resultados. Os *Pesos de Atenção*  $AW_{ij}$  obtidos após a aplicação da softmax são valores entre 0 e 1, e a soma de todos os pesos de atenção associados à **palavra de consulta** na posição  $i$  é igual a 1. Essa propriedade garante que a atenção seja distribuída corretamente entre todas as palavras na sequência, formando uma distribuição de probabilidade. Na Figura 131, o *Peso da Atenção* da palavra de consulta "anéis" para si própria é de 0.88 (88%), um peso elevado, como esperado. Já o *Peso da Atenção* que "anéis" deve dar para a palavra na Posição  $j = 4$  ("Saturno") é 0.10 (10%), e para a palavra na Posição  $j = 6$  ("belos") é 0.02 (2%), totalizando 1 ou 100% (em favor da simplicidade estamos assumindo que os *Pesos de Atenção* da **palavra de consulta** em relação às demais palavras da sentença são valores desprezíveis como 0,0001 por exemplo).

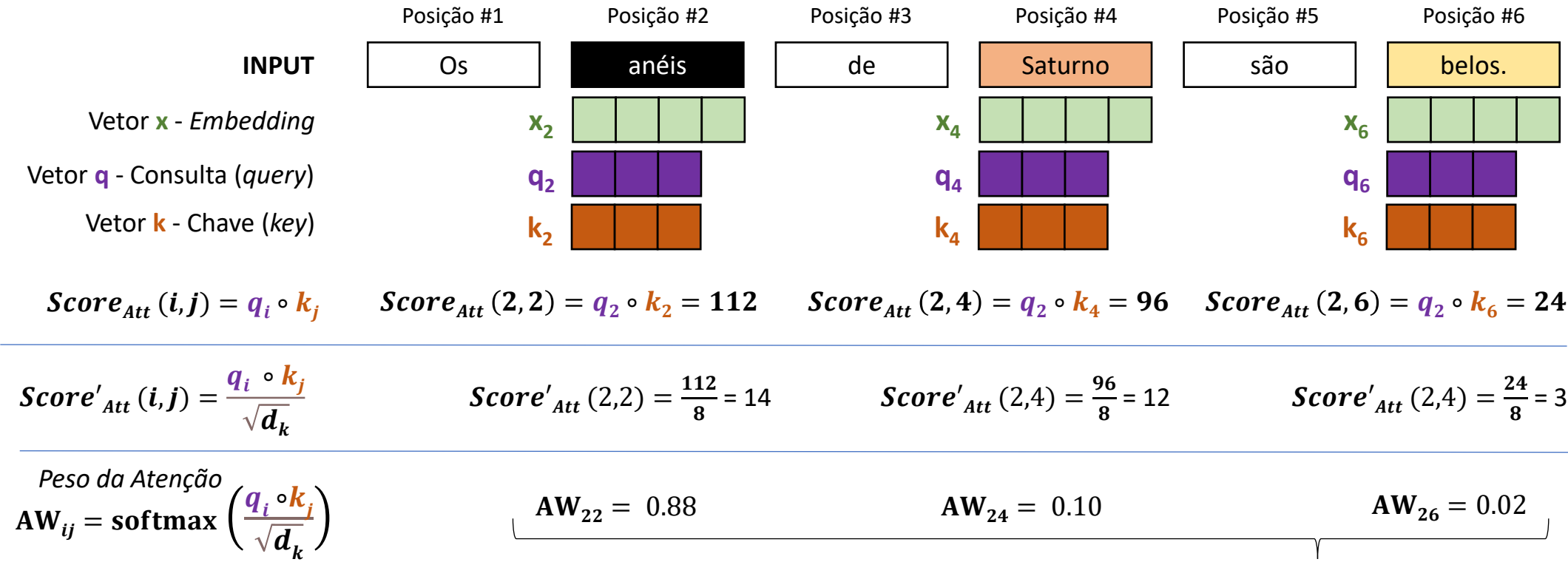


Figura 131 - Fonte: Adaptada de [105] pelo autor

$\sum_j AW_{ij} = 1 \text{ (100\%)}$

**Passo 4 - Saída da Atenção**

Finalmente, os **vetores de valor** ( $v_j$ ) de cada palavra na posição  $j$  são multiplicados (ponderados) pelos *Pesos de Atenção* (*Attention Weights* ( $AW_{ij}$ )) resultantes da função **softmax** para cada palavra  $j$  na sequência. Tecnicamente, é feita uma **combinação linear** dos vetores de valor  $v_j$  associados a cada palavra na sequência, produzindo um novo **vetor de atenção**  $z_i$ , único, associado com a palavra na **posição de consulta**  $i$ .

$$z_i = \sum_j AW_{ij} \times v_j$$

No exemplo, considerando apenas as palavras nas posições  $j=2, j=4$  e  $j=6$  e desprezando as demais,  $z_2$  será o **vetor de atenção** resultante da combinação linear de 3 vetores  $v_j$  ponderados pelos *Pesos de Atenção* ( $AW_{ij}$ ), sendo uma representação contextual da palavra de consulta em  $i = 2$ :

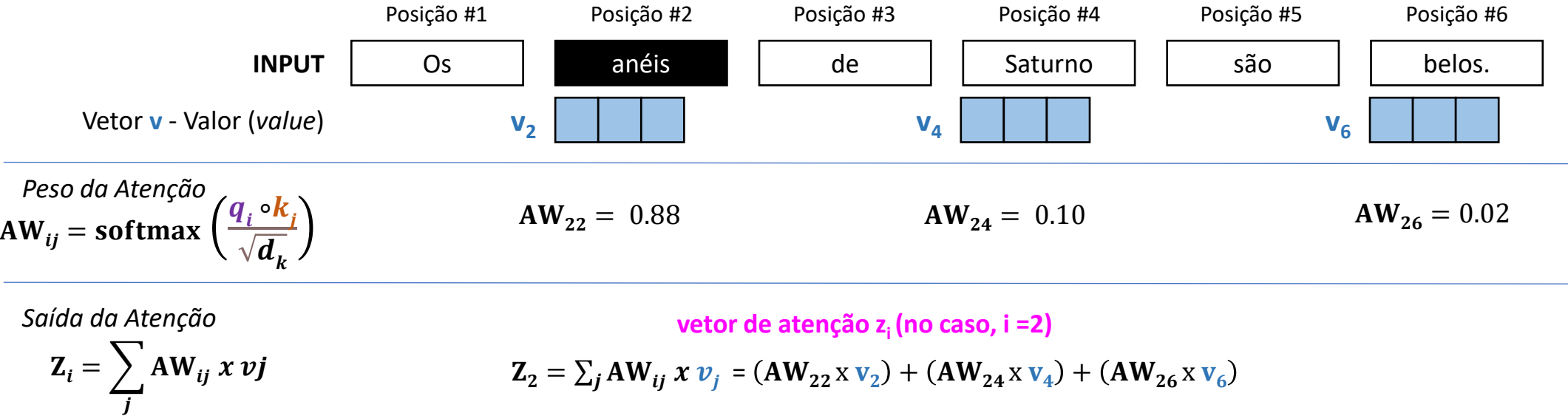


Figura 132 - Fonte: Adaptada de [105] pelo autor

## Resumindo:

O vetor de atenção  $\mathbf{z}_i$  representa o nível de atenção que a palavra na **posição de consulta  $i$**  deve dedicar a todas as palavras na sequência, incluindo ela própria. Esse vetor único é o resultado da ponderação dos **vetores de valor ( $\mathbf{v}_j$ )** associados a cada palavra na sequência pelos *Pesos de Atenção* ( $\mathbf{A}\mathbf{W}_{ij}$ ).

O vetor  $\mathbf{z}_i$  fornece uma representação contextualizada da palavra de consulta, levando em consideração a relevância relativa de todas as outras palavras na sequência (o quanto a informação de cada palavra é importante para a palavra em consulta). Isso permite que a arquitetura *Transformer* capture relações complexas e dependências semânticas em dados sequenciais de maneira eficaz.

No exemplo da Figura 133 espera-se que vetor  $\mathbf{z}_i$  indique que a **palavra em consulta** ("anéis") deve prestar muita atenção em si própria, também deve prestar atenção na palavra "Saturno" na posição  $j = 4$ , e um pouco menos de atenção na palavra "belos" na posição  $j = 6$ . Para efeitos práticos, a atenção da palavra "anéis" nas demais palavras da sentença foi considerada muito baixa e foi ignorada.

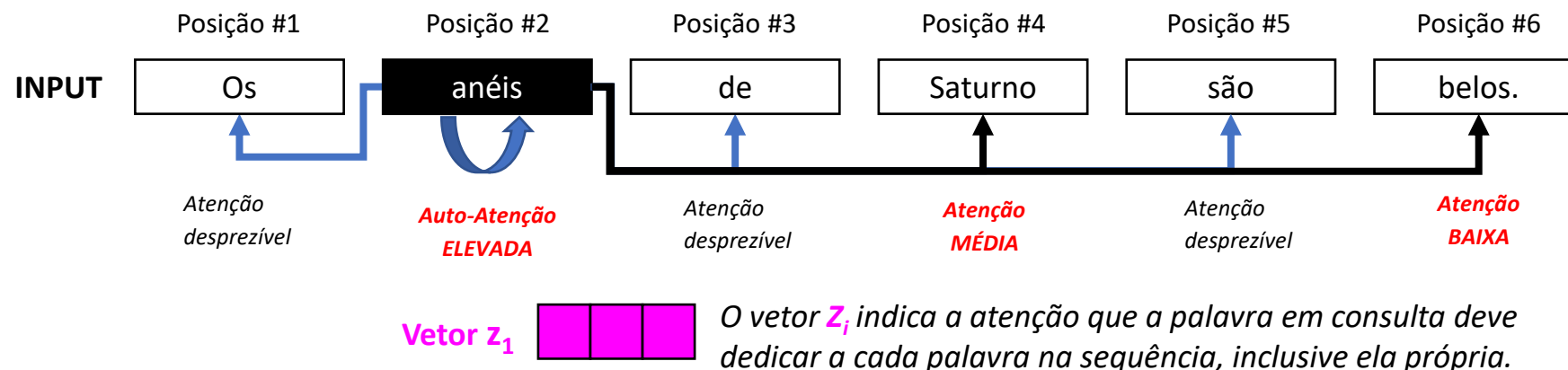


Figura 133 - Fonte: Adaptada de [105] pelo autor

Assim, a ponderação dos **vetores de valor  $v_j$**  pelos *Pesos de Atenção* ( $AW_{ij}$ ) gerados pela função **softmax** permite tanto "fortalecer" algumas palavras em certas posições da sentença que são mais importantes (merecem mais atenção por parte da palavra em consulta), quanto "enfraquecer" palavras irrelevantes (multiplicando seus **vetores de valor  $v_j$**  por pesos minúsculos como 0.001 por exemplo).

Vale lembrar que estes passos são repetidos para todas as palavras da sequência, permitindo que cada palavra seja colocada em posição de consulta e se calcule o quanto ela deve "prestar atenção" nela própria e em todas as outras palavras, com pesos determinados pelos cálculos explicados.

A Atenção permite que as redes *Transformer* capturem relações de proximidade (semântica, contexto) de longo alcance (mesmo entre palavras que estão distantes) em uma sequência - e isso faz enorme diferença para que o modelo tenha melhor desempenho em tarefas de tradução, ou de geração de texto ou outras.

#### 4.5.1.4. Matrizes em vez de vetores

Na realidade, em vez de fazer cálculos com vetores individuais ( $\mathbf{x}$ ,  $\mathbf{q}$ ,  $\mathbf{k}$ ,  $\mathbf{v}$ ), os cálculos de Auto-Atenção são feitos com matrizes  $\mathbf{X}$ ,  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  agregando vários vetores do mesmo tipo. Cada linha na matriz  $\mathbf{X}$  corresponde a um **vetor de Embedding** para uma palavra na sequência de entrada [106]. O produto da matriz  $\mathbf{X}$  pela matriz  $\mathbf{W}^Q$  gera a matriz  $\mathbf{Q}$ , o produto de  $\mathbf{X}$  por  $\mathbf{W}^K$  gera a matriz  $\mathbf{K}$ , e o produto de  $\mathbf{X}$  por  $\mathbf{W}^V$  gera a matriz  $\mathbf{V}$  (Figura 134).

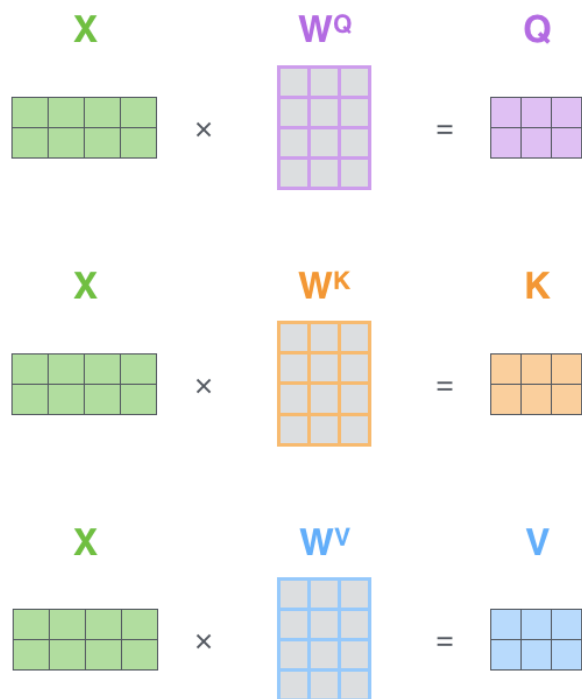


Figura 134 - Fonte: [105]

Como os cálculos são feitos com matrizes (no caso de  $\mathbf{K}$ , usa-se a matriz transposta,  $\mathbf{K}^T$ ), alguns dos passos anteriores mostrados com vetores para facilitar a compreensão do processo são na verdade realizados com maior eficiência, e o processo do cálculo da Auto-Atenção no ENCODER pode ser resumido em uma única fórmula reproduzida de [105]:

$$\text{softmax}\left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} = \mathbf{Z}$$

Figura 135 - Fonte: [105]

A Atenção resulta de multiplicações de matrizes e uma operação softmax.

A Figura 136 mostra Łukasz Kaiser (Research Scientist - Google Brain) explicando em [103] como a Atenção resulta da aplicação da função **softmax** ao produto escalar envolvendo as matrizes **Q** (**matriz de consultas**), **K<sup>T</sup>** (matriz *transposta* da matriz **de chaves**) e da **matriz de valores V**. Como vimos, a **matriz Q de consultas** é o "presente" (a palavra atual, que está sendo consultada), e as matrizes de **chaves** e **valores** são o "passado", uma "memória" de todas as palavras geradas anteriormente.

Em certo instante temos uma **palavra em consulta** (por exemplo, "anéis"). Com a palavra de consulta em foco, procuramos "no passado" as **chaves** mais semelhantes ou relacionadas a ela (como "Saturno").

De posse das **chaves**, obtemos os **valores** das palavras mais "similares" à palavra em consulta, e aplicamos a função **softmax** (exponenciação e normalização), obtendo uma "máscara", uma matriz com uma *Distribuição de Probabilidades sobre chaves*, onde os "picos" da distribuição são exatamente as palavras com maior similaridade (semântica ou contextual) com a palavra de consulta.

Multiplicamos a matriz com a *Distribuição de Probabilidades* pela **matriz V**, e temos a atenção final **Z**, ou seja, *as palavras que merecem mais atenção*.

Atenção: **Z** = A(**Q**,**K**,**V**) = **softmax** (**Q** **K<sup>T</sup>**) **V**

$$\text{softmax} \left( \frac{\begin{matrix} \text{Q} \\ \text{3x3 grid} \end{matrix} \times \begin{matrix} \text{K}^T \\ \text{3x3 grid} \end{matrix}}{\sqrt{d_k}} \right) \begin{matrix} \text{V} \\ \text{3x3 grid} \end{matrix} = \begin{matrix} \text{Z} \\ \text{3x3 grid} \end{matrix}$$

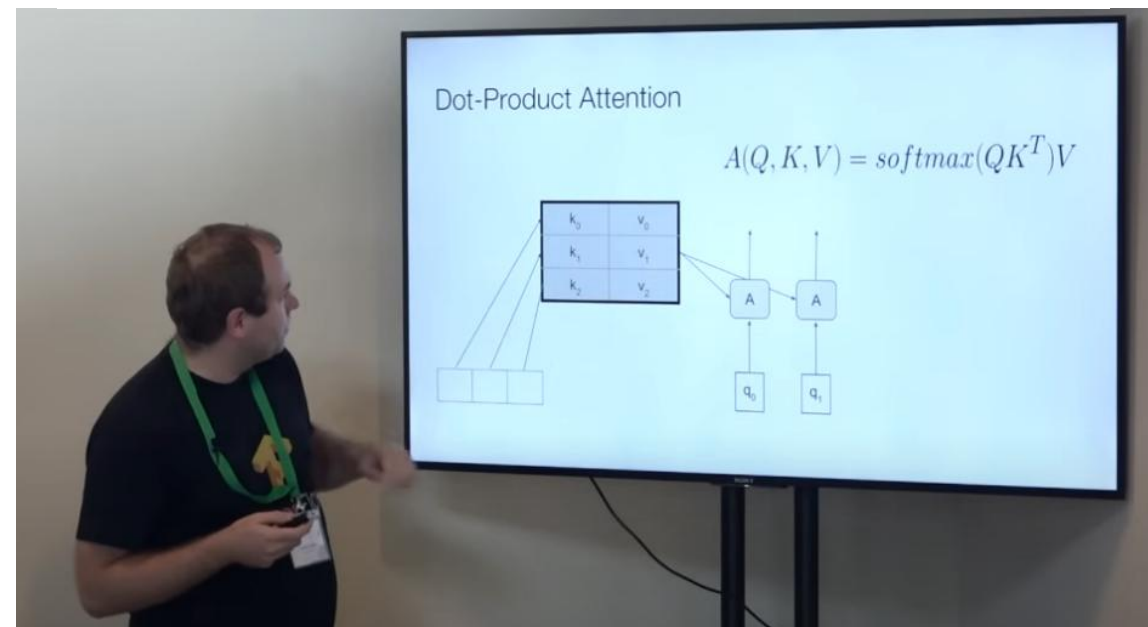


Figura 136 - Fontes: [103 e 105]

#### 4.5.1.5. A subcamada FFNN no ENCODER

Relembrando: O ENCODER #1 recebe uma lista de **vetores de Embedding  $x_i$**  como entrada, e processa esta lista passando os vetores (um de cada vez) para a subcamada de Auto-Atenção (*Self-Attention*). Para cada vetor  $x_i$  que entra na subcamada de Auto-Atenção é gerado um **vetor de Atenção  $z_i$**  que é então passado para a subcamada da rede neural FFNN (*Feed Forward Neural Network*). Por motivos didáticos a Figura 137 mostra vetores  $z_i$  individuais, mas na verdade, a camada FFNN recebe uma **matriz  $Z$**  com vários **vetores de atenção  $z_i$** , um para cada palavra da sequência.

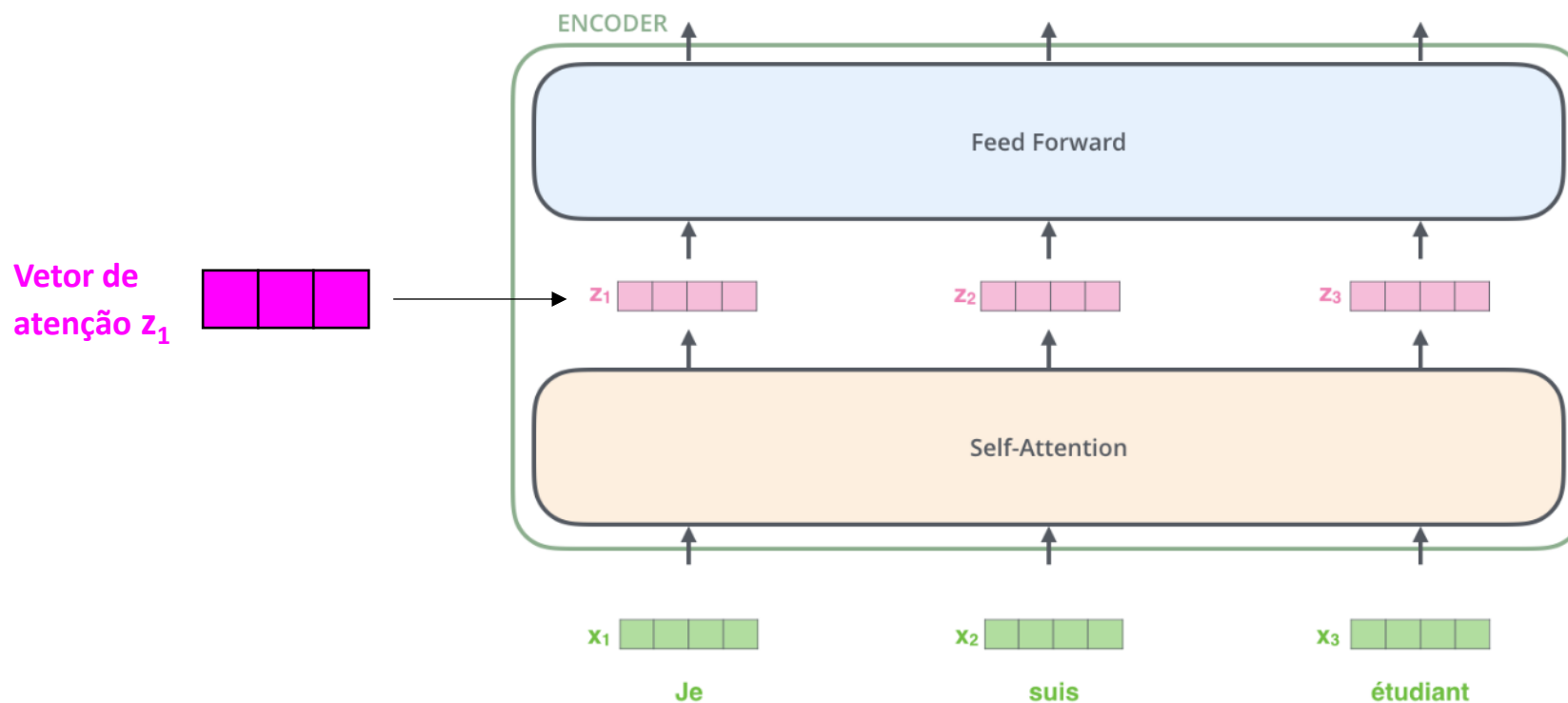


Figura 137 - Fonte: Adaptada de [105] pelo autor

O que acontece na subcamada *Feed Forward Neural Network (FFNN)* do ENCODER?

A subcamada FFNN age como um bloco de transformação, onde a entrada (matriz com **vetores de atenção  $z_i$** ) passa por uma operação linear seguida por uma não-linear, fornecendo representações mais expressivas e contextualizadas para cada posição na sequência. Essa capacidade de aprendizado de representações mais ricas permite que o modelo capture relações complexas e padrões não lineares nos dados, e é fundamental para o desempenho do *Transformer* em tarefas de processamento de linguagem natural e outras aplicações sequenciais.

Figura 138 - Cada posição na sequência, após passar pela subcamada de Auto-Atenção e pela subcamada FFNN, é representada por um **vetor de representação  $r_i$**  que contém informações contextualizadas e processadas, sendo a representação final do ENCODER # 1 para a palavra na posição  $i$  na sequência.

O vetor  $r_i$  (mais precisamente, a **matriz de representação  $R$**  contendo vários vetores  $r_i$ ) é passado para a próxima camada de ENCODER do Transformer até o final da pilha, e será a representação final de todas as palavras na sequência de entrada pelos ENCODERS.

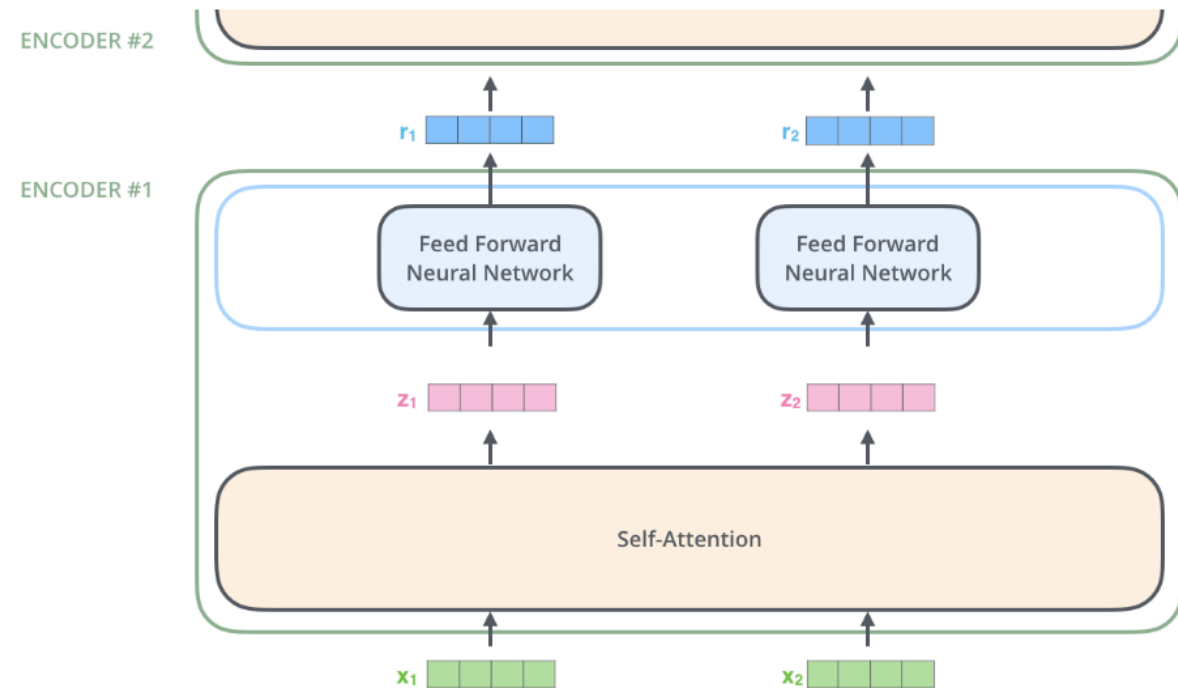


Figura 138 - Fonte: [105]



#### 4.5.1.6. Para que tantas camadas de ENCODERS?

Tal como definida no artigo "*Attention is all you need*" [80], a arquitetura *Transformer* tem seis camadas de ENCODERS. Ao utilizar múltiplas camadas, é aumentada a capacidade do modelo de aprender representações mais complexas e capturar padrões mais sutis em dados sequenciais. Isso permite que o modelo realize uma hierarquia de transformações e aprenda representações de níveis abstratos, o que pode ser fundamental para tarefas que exigem uma compreensão mais profunda da linguagem natural.

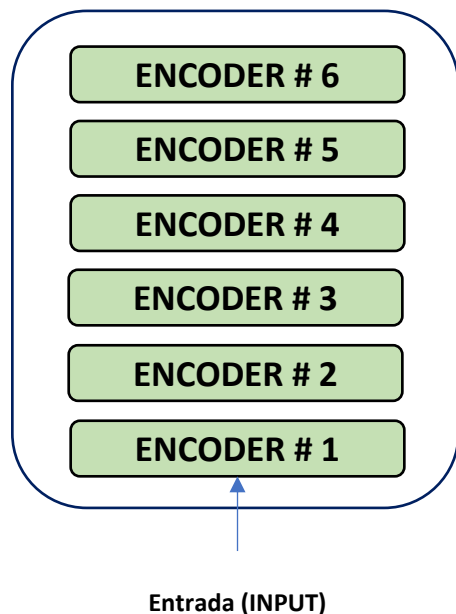


Figura 139 - Fonte: Adaptado de [105]

Considere uma tarefa de análise de sentimento em textos. Suponha que você tenha uma sentença como "O filme foi bom, mas a trilha sonora deixou a desejar." Aqui estão alguns níveis de representação que poderiam ser aprendidos por diferentes camadas de ENCODERS:

- A primeira camada pode aprender **representações básicas** de palavras e suas interações imediatas. Por exemplo, pode capturar que "filme" e "bom" estão associados a um sentimento positivo, enquanto "trilha sonora" pode estar associada a um sentimento negativo.
- A segunda camada pode combinar essas representações básicas para **aprender representações mais complexas**. Pode capturar a conjunção "mas" e entender que, apesar do sentimento positivo inicial, a frase seguinte modifica essa emoção.
- A terceira camada pode aprender a **hierarquia de contextos**, compreendendo que a qualidade da trilha sonora pode ser contextualizada pela expressão "deixou a desejar" após a conjunção "mas".
- Camadas posteriores podem capturar relações de longo alcance e dependências semânticas mais complexas. Por exemplo, podem entender que a palavra "deixou a desejar" é crítica para a avaliação do sentimento geral da sentença.

#### 4.5.1.7. Atenção Multi-Head

Observando o diagrama da arquitetura (Figura 140), podemos notar que as subcamadas de Auto-Atenção (*Self Attention*) em cada ENCODER estão na verdade reunidas em um componente chamado **Multi-Head Attention** ("Atenção com Múltiplas Cabeças").

Em termos simples, isso significa que não há apenas "uma" Atenção, mas várias ocorrendo simultaneamente. No artigo que apresenta a arquitetura original [80] são oito "cabeças de atenção" para cada bloco de Atenção no ENCODER, e mais oito para cada bloco no DECODER. Isso permite criar *em paralelo* diferentes "representações" das sequências em análise. Em vez de lidar com apenas um grupo de matrizes de pesos de consulta, chave e valor (**Q**, **K**, **V**) de cada vez, a rede trabalha com vários conjuntos de matrizes ao mesmo tempo.

Cada conjunto de "matrizes de Atenção" é inicializado de forma aleatória no início do processo de treinamento (os pesos recebem valores aleatórios) [105]. Após o treinamento, cada conjunto é utilizado para projetar os **vetores x de Embeddings** em um "espaço de representação" diferente.

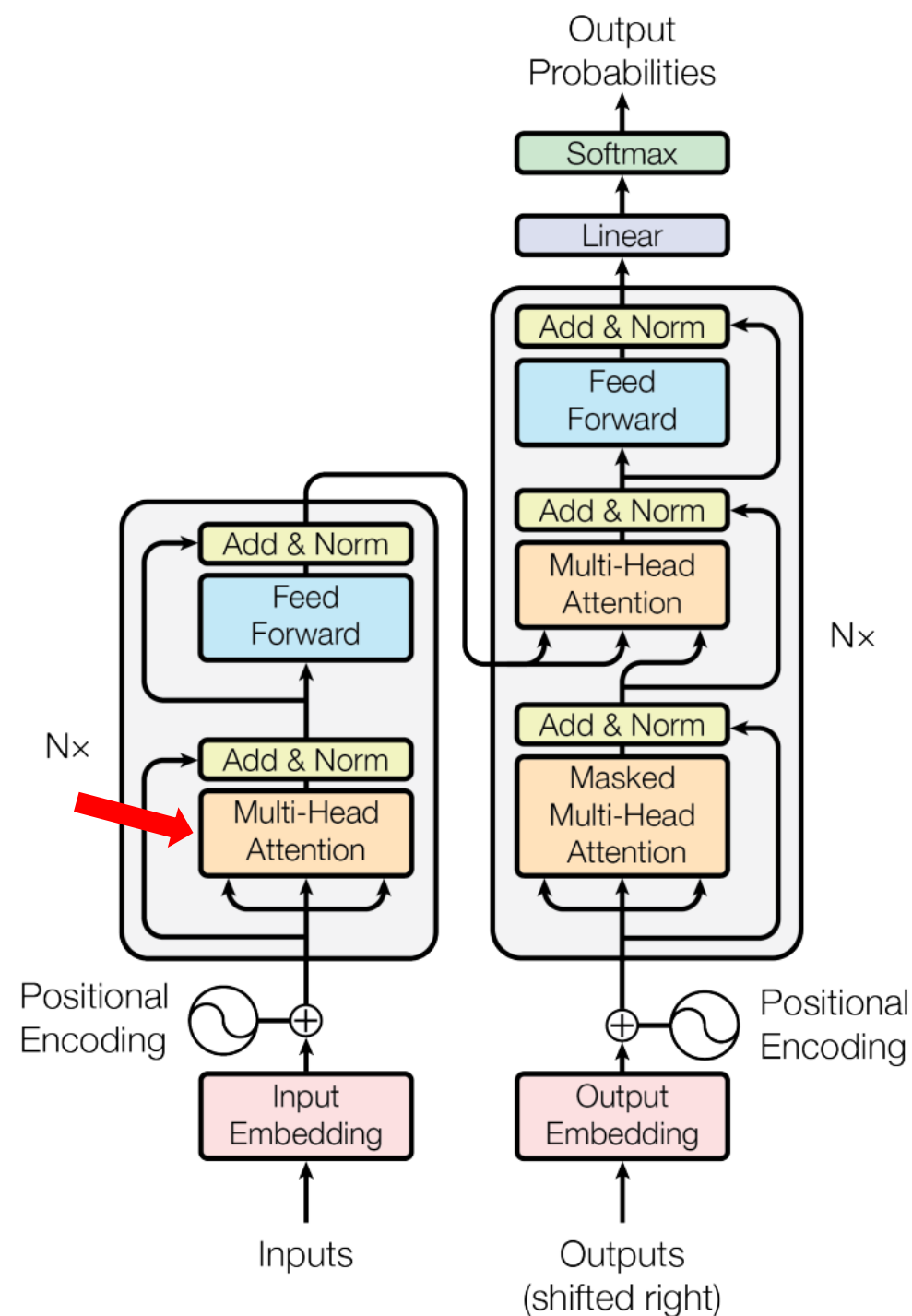


Figura 140: Fonte: [80]

Para simplificar, a Figura 141 mostra *apenas duas* (das oito) cabeças de Atenção (*Attention Heads*) previstas na arquitetura original (numeradas de AH #0 até AH #7). Em cada uma, a cada instante, haverá uma representação diferente da *matriz X* contendo *vetores x de Embedding* de uma sequência qualquer (no exemplo de [105] a sequência tem apenas dois vetores, cada um com uma palavra da frase "*Thinking Machines*", ou "Máquinas Pensantes").

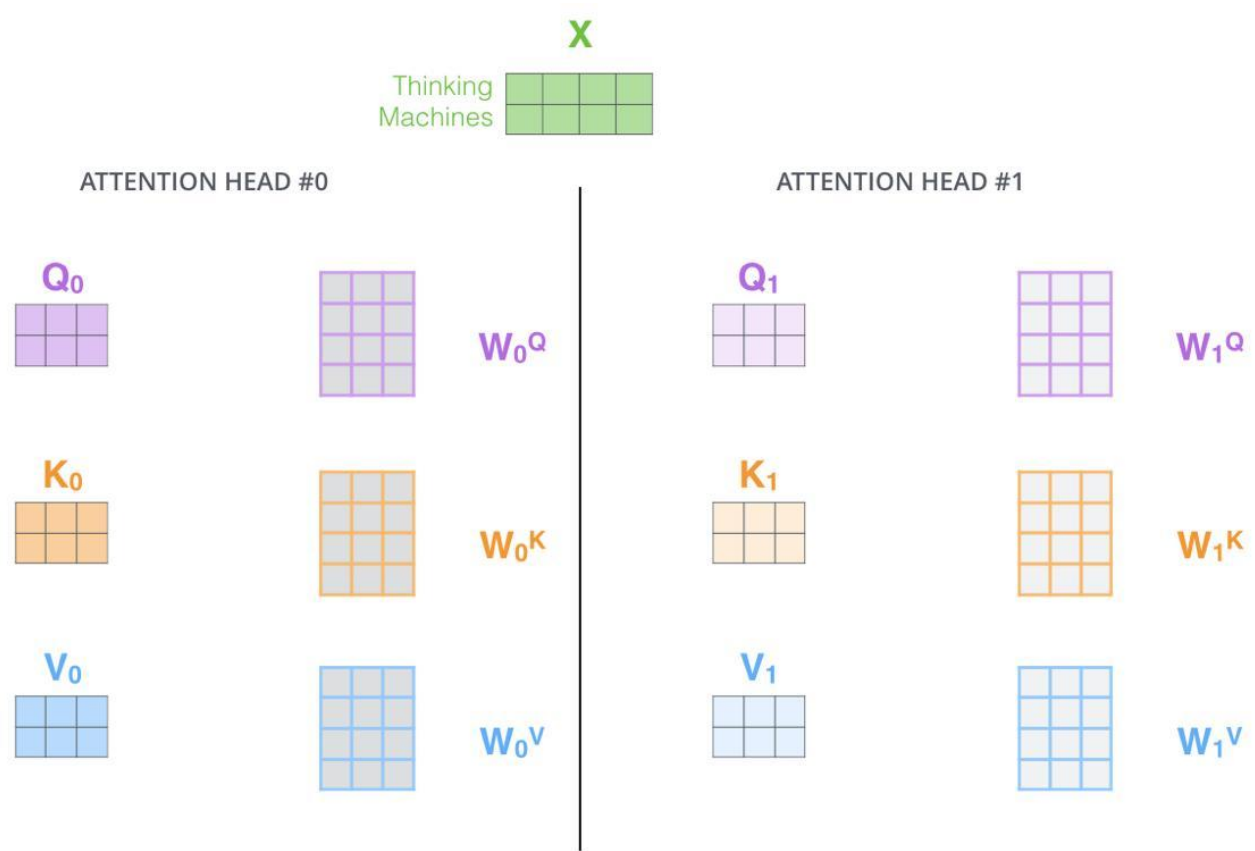


Figura 141 - A Atenção Multi-Headed mantém matrizes de pesos ( $W_j^Q$ ,  $W_j^K$ ,  $W_j^V$ ) separadas para cada "cabeça de Atenção". Em uma das oito "cabeças" de índice  $j=0$  até  $j=7$ , a matriz **X** contendo os *vetores de Embeddings* é multiplicada pelas matrizes de pesos da cabeça  $j$  ( $W_j^Q$ ,  $W_j^K$ ,  $W_j^V$ ) para gerar as matrizes de consulta, chave e valor  $Q_j$ ,  $K_j$ ,  $V_j$ . Por exemplo, na cabeça de Atenção #1 ( $j = 1$ ), o produto da matriz **X** pela matriz  $W_1^Q$  gera a matriz  $Q_1$ , o produto de **X** por  $W_1^K$  gera a matriz  $K_1$ , e o produto de **X** por  $W_1^V$  gera a matriz  $V_1$ .

Figura 141 - Fonte: [105]

Imagine agora que todos os cálculos de Auto-Atenção que discutimos agora são executados não em uma, mas em oito cabeças de Atenção, cada uma com matrizes de pesos diferentes. Em vez de obter uma única **matriz Z**, contendo os **vetores de atenção  $z_i$**  para diferentes partes da sequência, agora teremos oito **matrizes Z** (denotadas pelos índices de  **$Z_0$**  até  **$Z_7$**  na Figura 142).

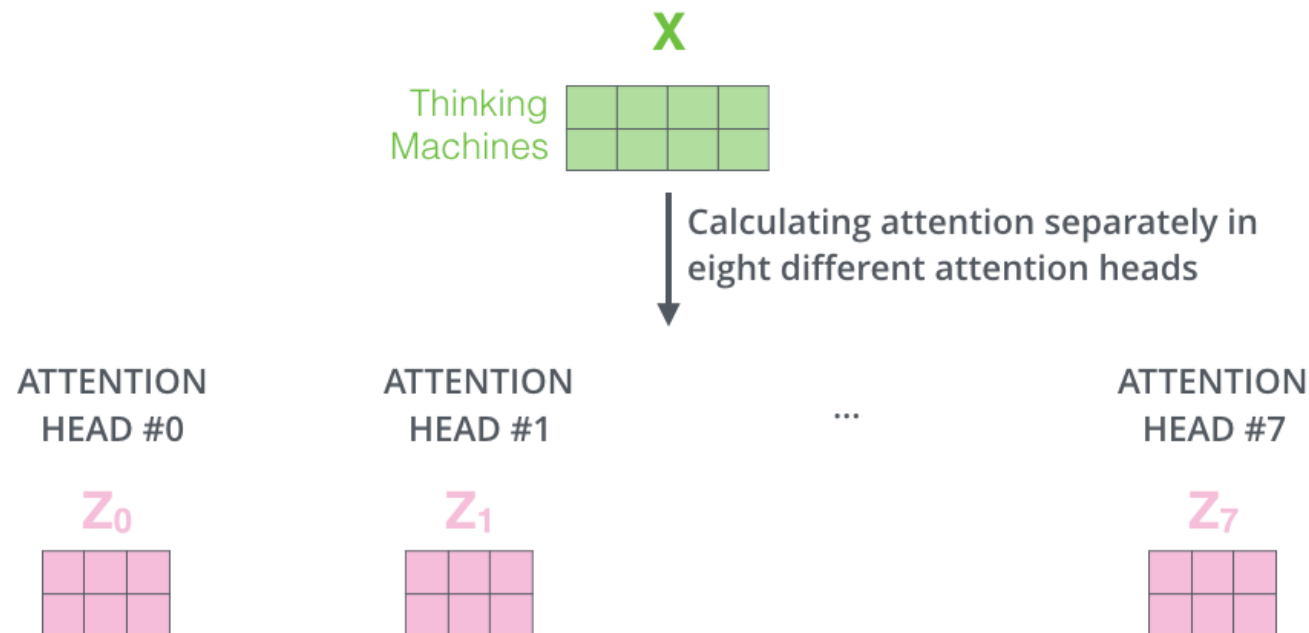
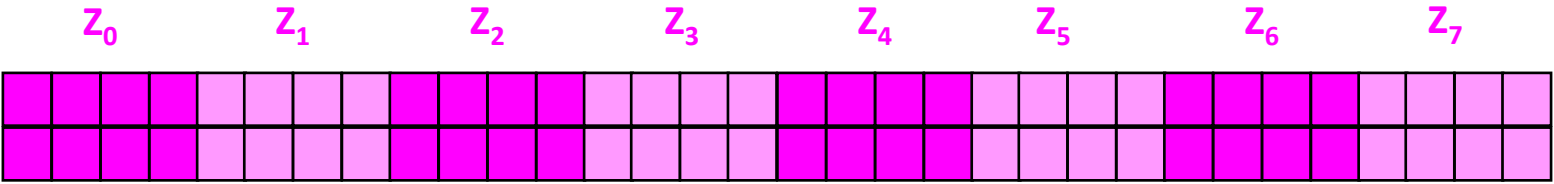


Figura 142 - Fonte: [105]

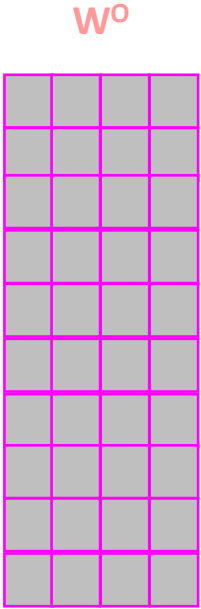
Porém, a subcamada FFNN (Rede Neural *Fast-Forward*) em cada ENCODER espera receber uma única **matriz de atenção Z** (composta por diferentes vetores  **$z_i$** , um para cada palavra), e não oito delas. Como resolver isso?

Jay Alamar nos explica [105] que para que a camada FFNN receba uma matriz de atenção unificada, as oito matrizes resultantes das oito cabeças de Atenção são concatenadas em uma matriz única. Em seguida, esta **matriz de Atenção Z** unificada é multiplicada por uma nova matriz de pesos  $W^O$ , que é treinada com o modelo. O resultado é uma **matriz Z** que unifica as informações de todas as cabeças de Atenção, e que será enviada para a subcamada FFNN (Figura 143).

(1) As **Matrizes Z** das 8 cabeças de Atenção são concatenadas em uma matriz única...



... (2) que é multiplicada por uma matriz de pesos  $W^O$  cujos pesos são atualizados durante o treinamento



(3) O resultado é uma **matriz de Atenção única Z**, que é passada para a subcamada FFNN do ENCODER.

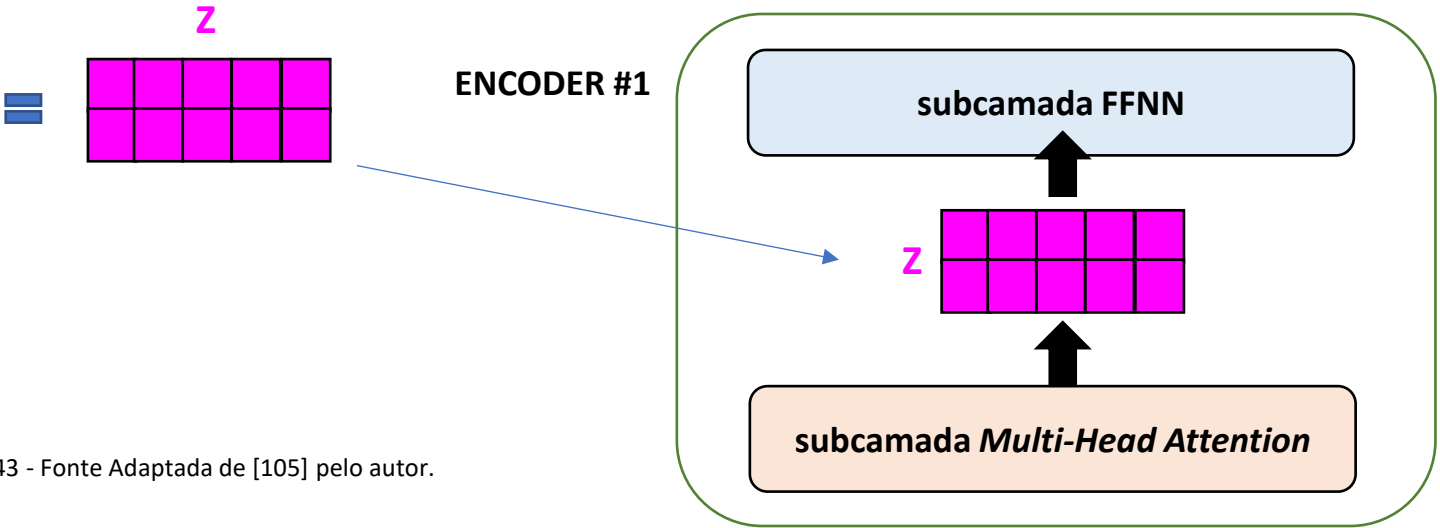


Figura 143 - Fonte Adaptada de [105] pelo autor.

O diagrama contém um resumo da geração da **matriz de atenção unificada Z**, que será passada para a subcamada FFNN acima, dentro do ENCODER.

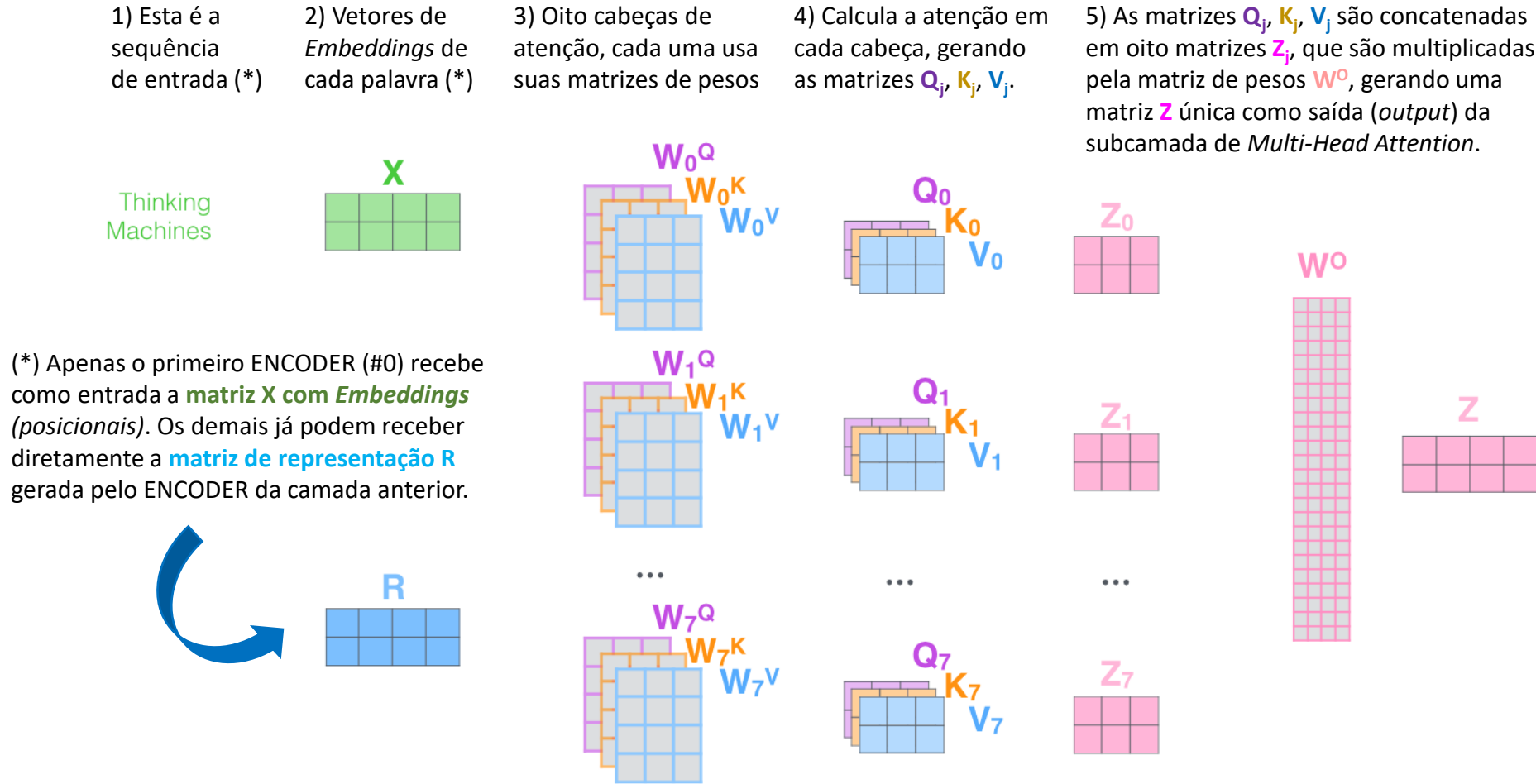


Figura 144 - Fonte Adaptado de [105] pelo autor.

## PRIMEIRO ENCODER #1

Observando a Figura 144 com atenção, notamos que há uma pequena diferença na entrada (*Input*) do primeiro ENCODER da pilha, em relação aos demais ENCODERS. Apenas o primeiro ENCODER recebe como entrada uma **matriz X com os vetores de Embeddings**, e gera com saída a sua **matriz de representação R** com a representação final da sequência de entrada, gerada pela subcamada FFNN (Figura 145).

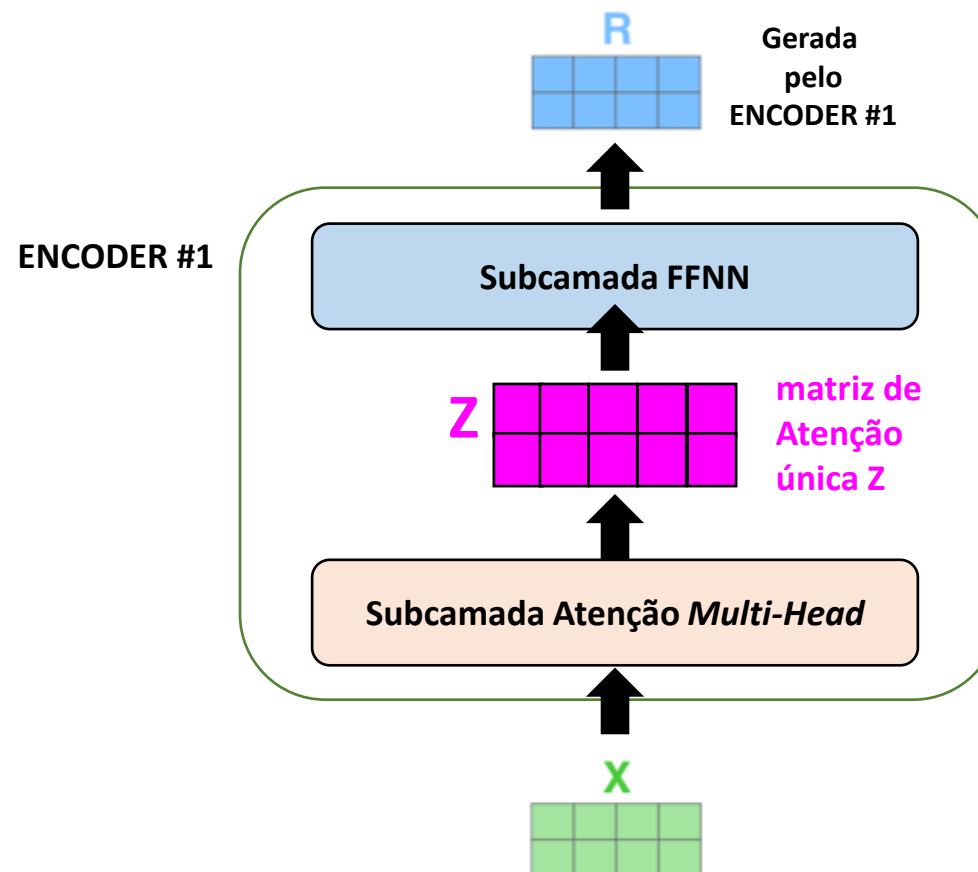


Figura 145 - Fonte Adaptado de [105] pelo autor.

## ENCODERS #2 até #6

Os demais cinco ENCODERS (considerando a pilha de seis da especificação original) não precisam mais receber **vetores de Embeddings**, pois já podem receber a **matriz de Representação R** gerada pela subcamada FFNN (*Feed Forward Neural Network*) do ENCODER anterior.

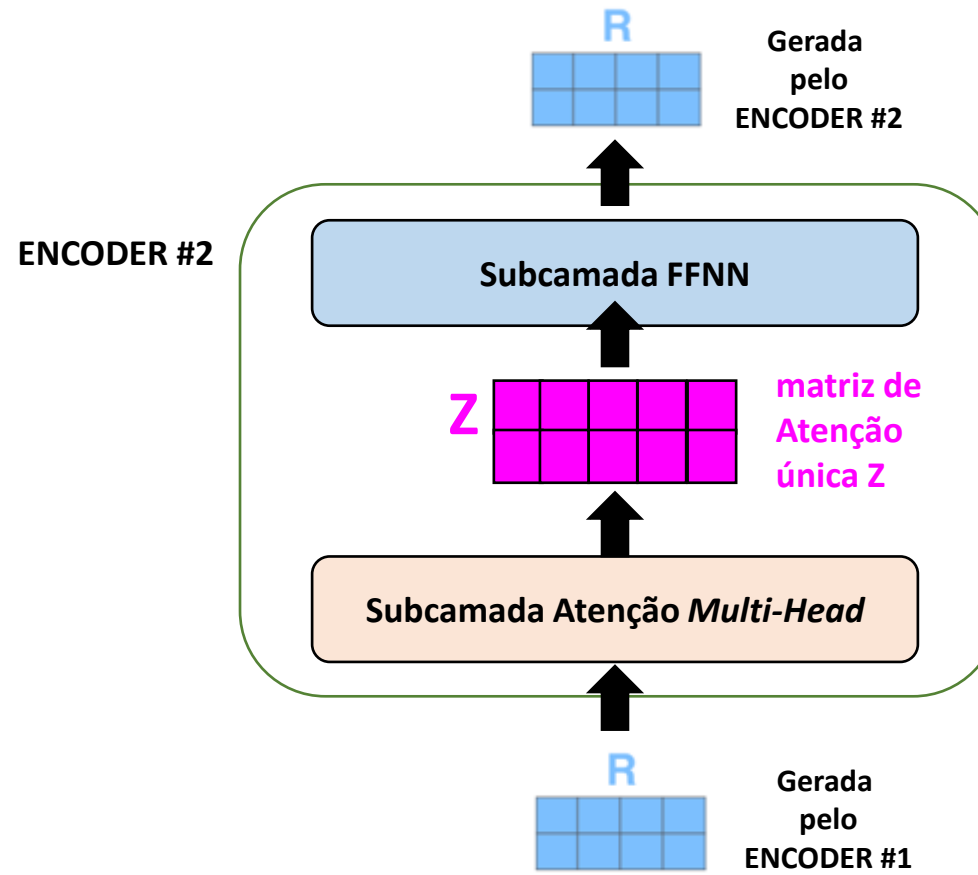


Figura 146 - Fonte Adaptado de [105] pelo autor.



Vamos revisitar agora um exemplo que já tínhamos discutido. A Figura 147 mostra com cores onde as oito cabeças de Atenção estão focadas. Enquanto a palavra "it\_" (que no caso, em Português equivale ao pronome "ele\_", que substitui "o animal") está sendo processada, uma cabeça de Atenção está mais focada em "O animal", enquanto outra está focando em "tired\_" ("cansado\_"). De alguma forma, a rede captou a relação entre as palavras "animal", "ele" e "cansado", embora estejam bem distantes na sequência.

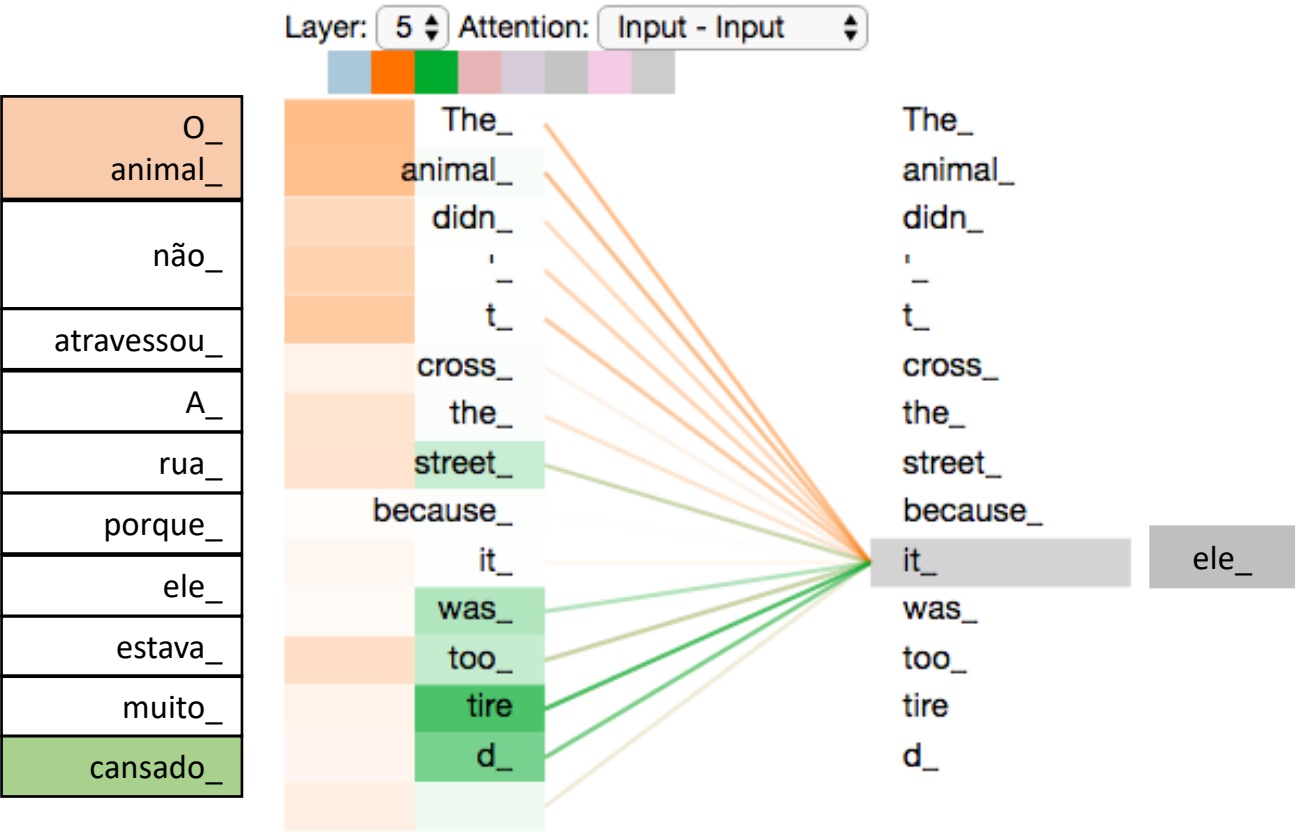


Figura 147 - Fonte: Adaptada de [105] pelo autor

#### 4.5.1.8. Conexões residuais e camada de normalização

Observando a arquitetura *Transformer*, podemos notar que cada subcamada de Auto-Atenção e de FFNN (tanto no ENCODER quanto no DECODER) é seguida de uma camada de soma e normalização (**Add & Norm**), em amarelo, destacada pelas setas na Figura 148.

Como o nome sugere, essa etapa é composta por duas operações principais: uma adição (**Add**) seguida de uma normalização (**Norm**). Aqui, surge um novo conceito, o de "conexão residual", onde a saída de uma subcamada (Auto-Atenção ou *Feed Forward*) é somada à entrada original dessa mesma subcamada (BOX 9).

Esse processo é repetido para cada subcamada dentro do ENCODER e do DECODER na arquitetura *Transformer*.

Mas o que está sendo somado, e para que serve a normalização?

Vamos dar um "zoom" neste processo para entendê-lo melhor.

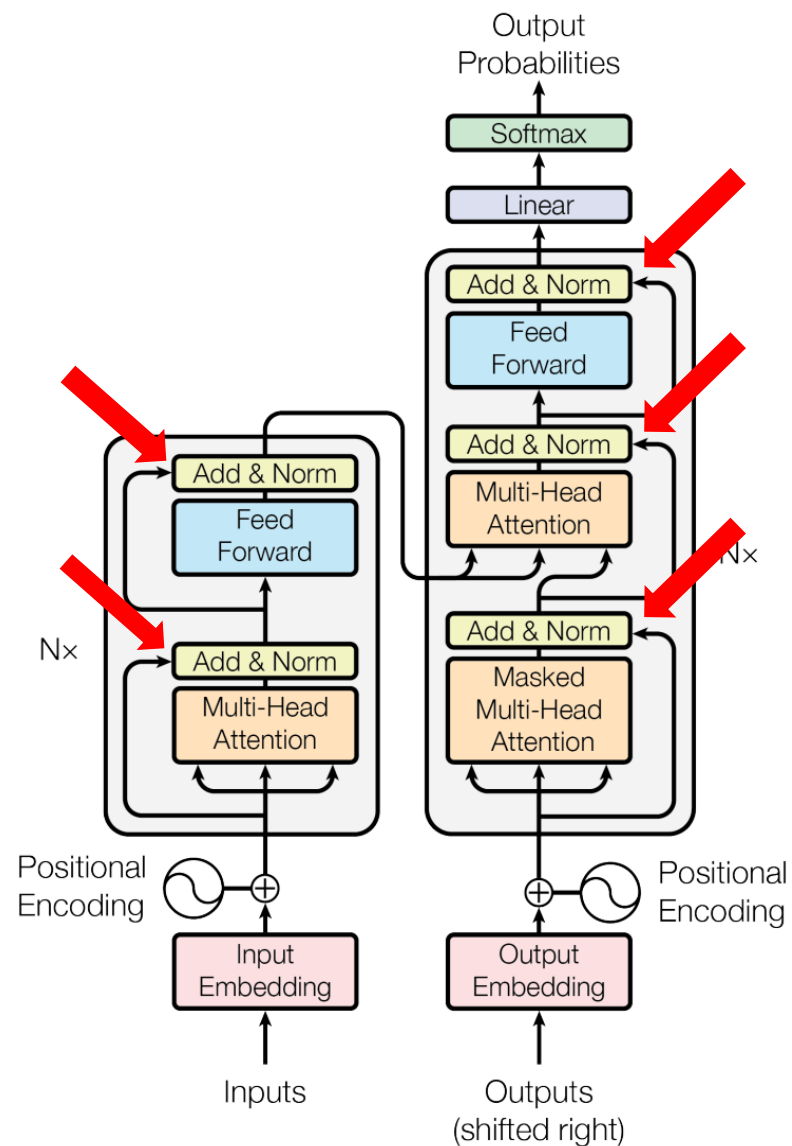


Figura 148: Fonte: [80]

## BOX 9. Conexão residual

Em cada ENCODER e DECODER, a saída de uma subcamada (Auto-Atenção ou da camada *Feed Forward*) é somada à entrada original dessa mesma subcamada. Esta soma é chamada de **conexão residual** (*residual connection*). A conexão residual é uma técnica usada para facilitar o treinamento de redes neurais com muitas camadas (*Deep Learning*).

Considere uma camada genérica em uma rede neural, representada por uma função  $H(\mathbf{x})$ , onde  $\mathbf{x}$  é a entrada e  $H(\mathbf{x})$  é a saída após a aplicação das operações da camada (por exemplo, uma camada de Auto-Atenção, ou uma camada de *Feed Forward* como as utilizadas na arquitetura *Transformer*). A operação de conexão residual em uma camada pode ser expressa por: **Output = input +  $H(\text{input})$** , onde **Input** é a **entrada original** da camada e  **$H(\text{Input})$**  é a transformação aplicada a essa entrada pela camada. A saída final é a soma da entrada original e da transformação realizada.

A principal motivação por trás das conexões residuais é facilitar o treinamento de redes profundas (redes neurais com muitas camadas). Durante o treinamento, as redes neurais aprendem a ajustar os pesos para minimizar a diferença entre a saída desejada e a saída prevista (ou seja, minimizar uma função de erro). No entanto, em redes com muitas camadas, pode ocorrer o problema de *Vanishing Gradients* (gradientes que ficam muito pequenos à medida que são propagados para trás na rede) que discutimos na Seção 4.2. Este problema pode tornar o treinamento mais difícil, pois os pesos nas camadas anteriores não são ajustados de forma adequada. A conexão residual fornece um caminho direto para o gradiente fluir da saída diretamente para a entrada *da mesma camada*. Isso facilita a propagação do gradiente durante o treinamento, ajudando a mitigar o problema dos gradientes que desaparecem.

Na arquitetura *Transformer*, essa conexão residual é implementada adicionando (**Add**) a entrada original diretamente à saída da subcamada (de Auto-Atenção ou de *Feed Forward*), antes de passar pela camada de normalização (**Norm**). O resultado é que a informação original é preservada e usada para ajustar os pesos durante o treinamento.

A Figura 149 dá um "zoom" no processo de conexão residual e normalização para a subcamada de Auto-Atenção (*Multi-Head Attention*) de um ENCODER (no caso, o primeiro da pilha, já que recebe como entrada **vetores x de Embeddings**). Como já explicado, na camada **Add & Norm** há uma soma das matrizes **X** e **Z**, seguida de uma normalização.

**Add:** Note que a **matriz X** é a matriz que entra, e a **matriz Z** é a matriz que sai da subcamada de Auto-Atenção (*Self-Attention*). Então, o que entra na camada está sendo somado ao que sai da mesma camada, o que caracteriza uma **conexão residual** tal como explicado no BOX 10.

**Norm:** O resultado da soma anterior é normalizado pela **camada de normalização** onde é realizada uma transformação linear que normaliza a média e o desvio padrão da saída.

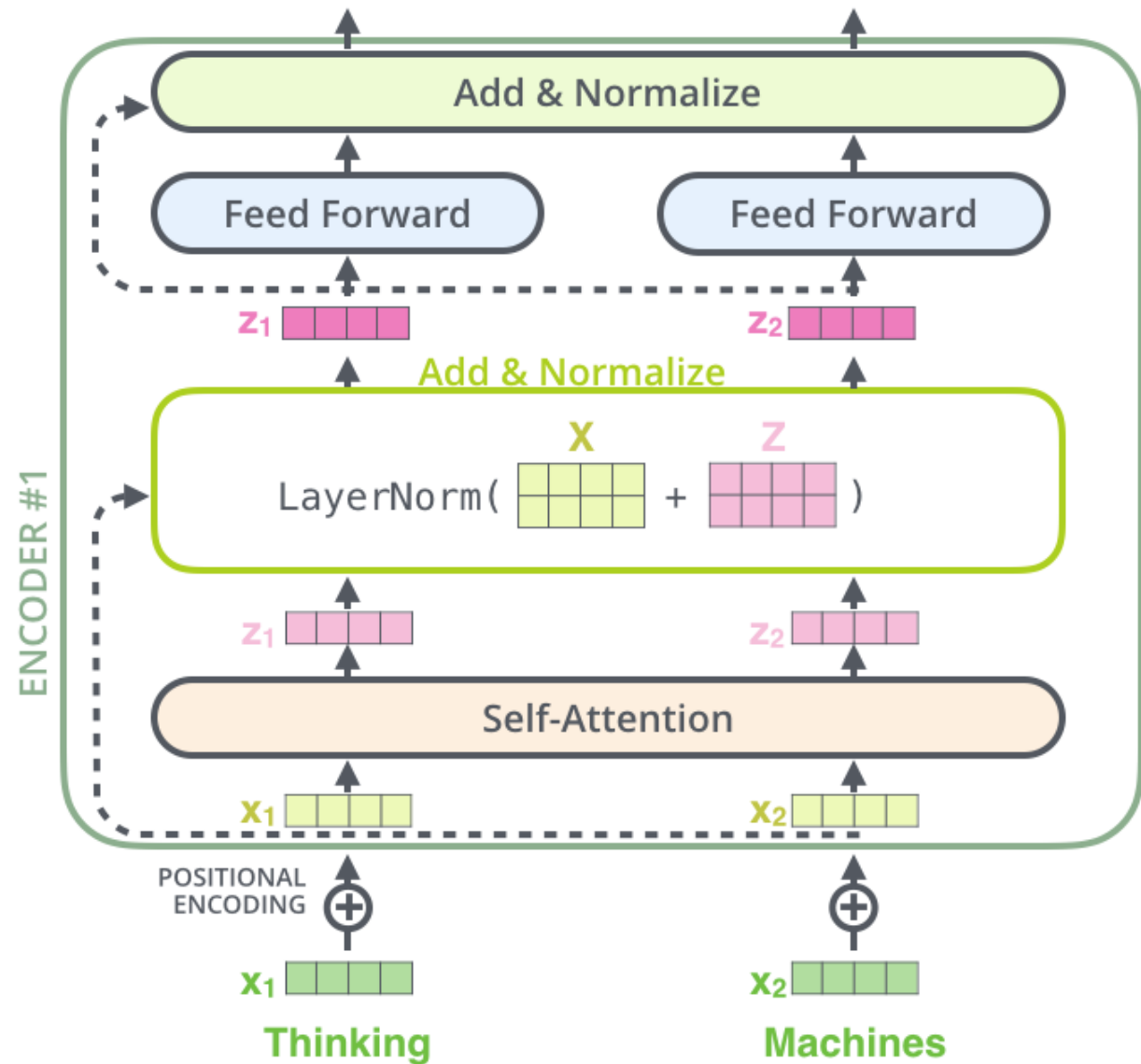


Figura 149: Fonte: [105]

O processamento de uma sequência de entrada (*input*) atravessa a pilha de ENCODERS (seis, na arquitetura original de [80]). A saída da última camada do último ENCODER é uma *matriz de Representação R* final, que contém informações sobre todas as palavras na sequência de entrada, incluindo (da melhor forma possível) as suas relações semânticas.

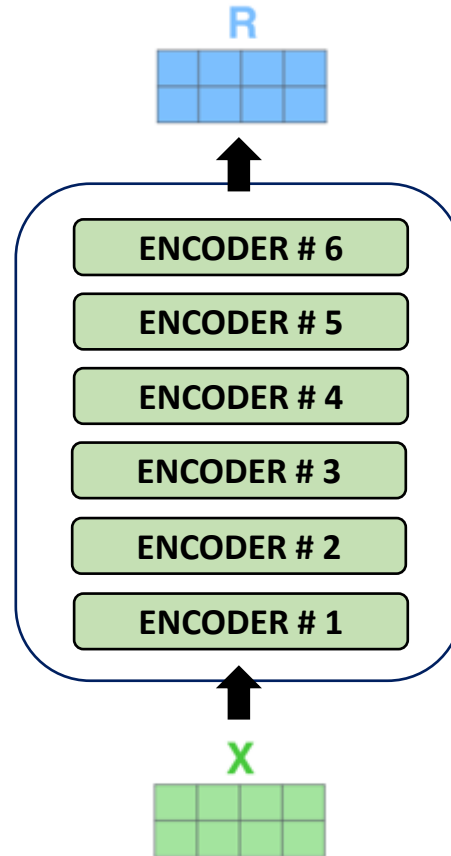


Figura 150 - Fonte Adaptado de [105] pelo autor.

Isso encerra nossa discussão sobre os ENCODERS. Vamos tratar dos DECODERS em seguida.

## 4.5.2. DECODERS

Muitos dos conceitos que aprendemos nas discussões sobre ENCODERS também são aplicáveis aos DECODERS, mas há algumas diferenças. O primeiro ponto a ressaltar é que nos DECODERS, as subcamadas de Auto-Atenção são mascaradas (*Masked Multi-Head Attention*), de modo que só "prestam atenção" em posições *anteriores* da sequência geradas como saídas do próprio DECODER. Além disso, em cada DECODER há uma subcamada adicional, a **Atenção Encoder-Decoder** (*Encoder-Decoder Attention*), que ajuda o DECODER a "prestar mais atenção" nos locais corretos da sequência de entrada. A Figura 151 é uma simplificação da arquitetura original [80].

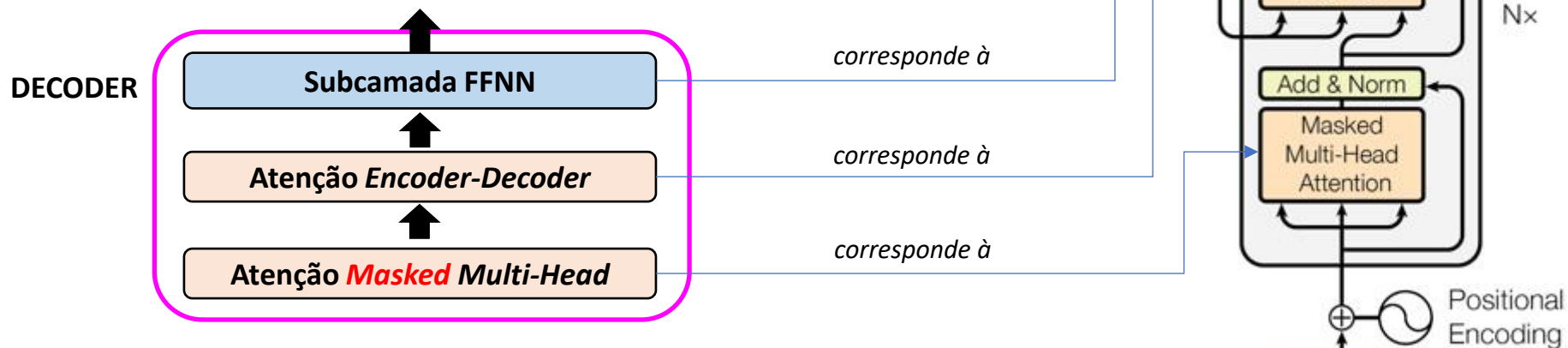
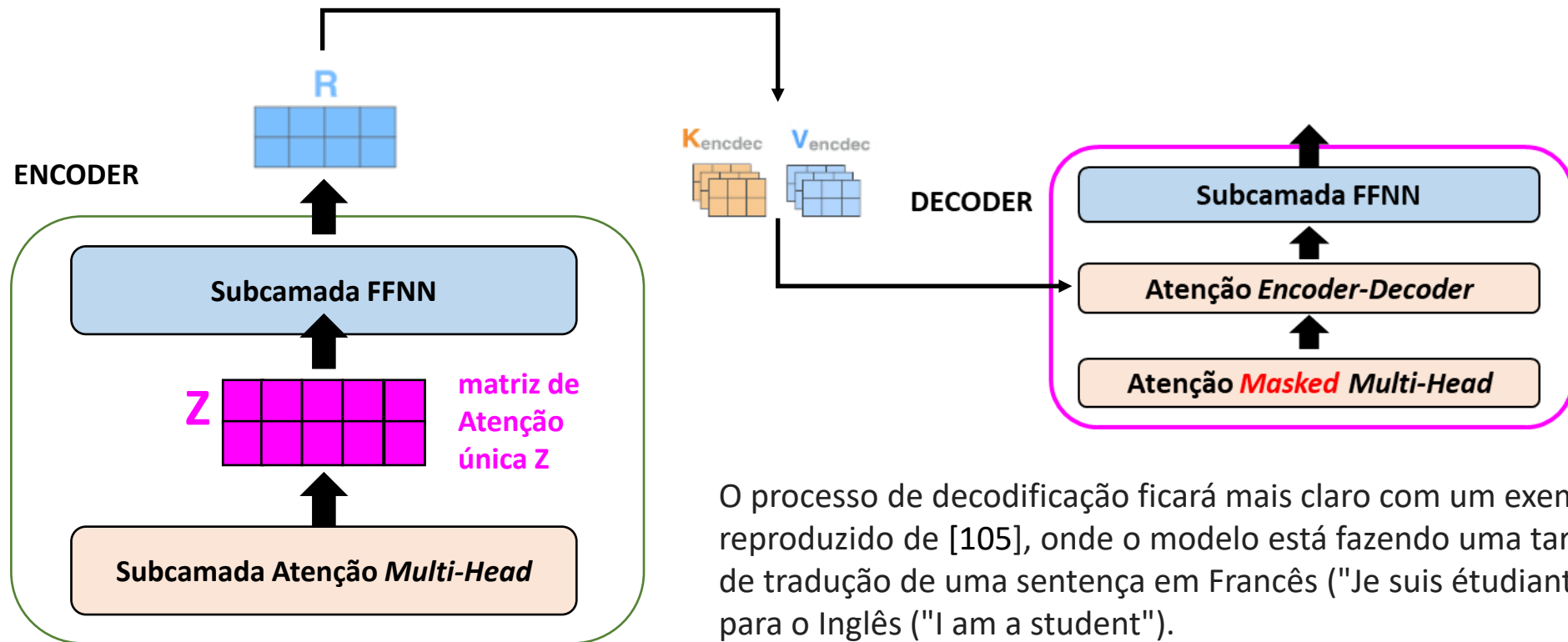


Figura 151: Fonte: Adaptada de [105] pelo autor

Figura 151 - No DECODER, a Auto-Atenção Multi-Head (*Masked Multi-head Attention*) é mascarada para evitar que uma posição na sequência acesse informações futuras durante a predição. Isso é feito aplicando uma máscara na Matriz de Atenção que bloqueia conexões para posições subsequentes na sequência. Assim, cada posição na sequência pode atentar apenas para *tokens* anteriores ou para si mesma.

A **matriz de Representação R** gerada como saída final pela pilha de ENCODERS é usada para gerar as matrizes **chave (K)** e **valor (V)**, ainda na etapa de codificação (BOX 10). Essas duas matrizes são utilizadas na subcamada de Atenção Encoder-Decoder (*Encoder-Decoder Attention*) no DECODER, durante a decodificação (Figura 152). Como mencionado, a subcamada extra de Atenção Encoder-Decoder ajuda o DECODER a decidir quais partes da sequência de entrada são mais importantes para *gerar a próxima palavra na sequência de saída* (como a próxima palavra em uma tradução).



O processo de decodificação ficará mais claro com um exemplo reproduzido de [105], onde o modelo está fazendo uma tarefa de tradução de uma sentença em Francês ("Je suis étudiant") para o Inglês ("I am a student").

Figura 152: Fonte: Adaptada de [105] pelo autor

## BOX 10. Representação final R e vetores K e V

Lembre-se da discussão sobre os ENCODERS que a camada de **Atenção Multi-Head** (*Multi-Head Attention*) calcula a Atenção em cada cabeça gerando três matrizes de pesos: A **matriz de Consultas Q**, a **matriz de Chaves K**, e a **matriz de Valores V**. Como existem oito cabeças de Atenção (na especificação original), são geradas oito matrizes de cada tipo. As matrizes  $Q_j$ ,  $K_j$  e  $V_j$  são concatenadas (unificadas) em oito matrizes  $Z_j$ , que são multiplicadas pela matriz de pesos  $W^O$ , gerando uma **matriz Z** única como saída (*output*) da subcamada de *Multi-Head Attention*. Considerando uma pilha de seis ENCODERS, no último da pilha (#6) gera a **matriz de Representação R** final.

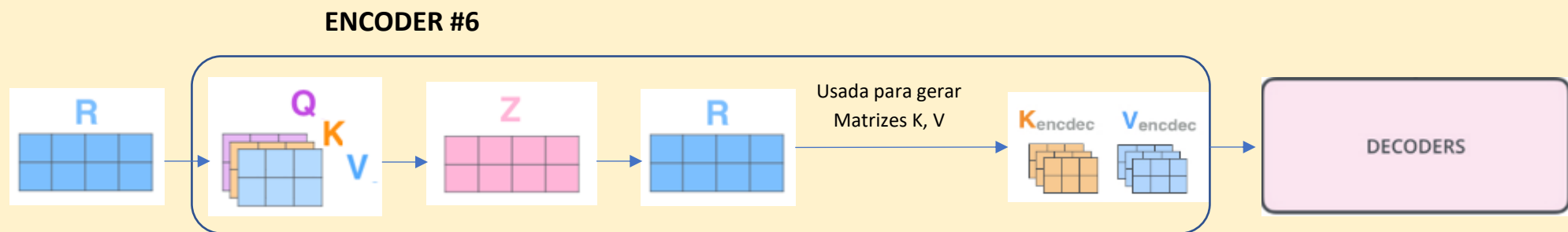


Figura 153: Fonte: Adaptada de [105] pelo autor

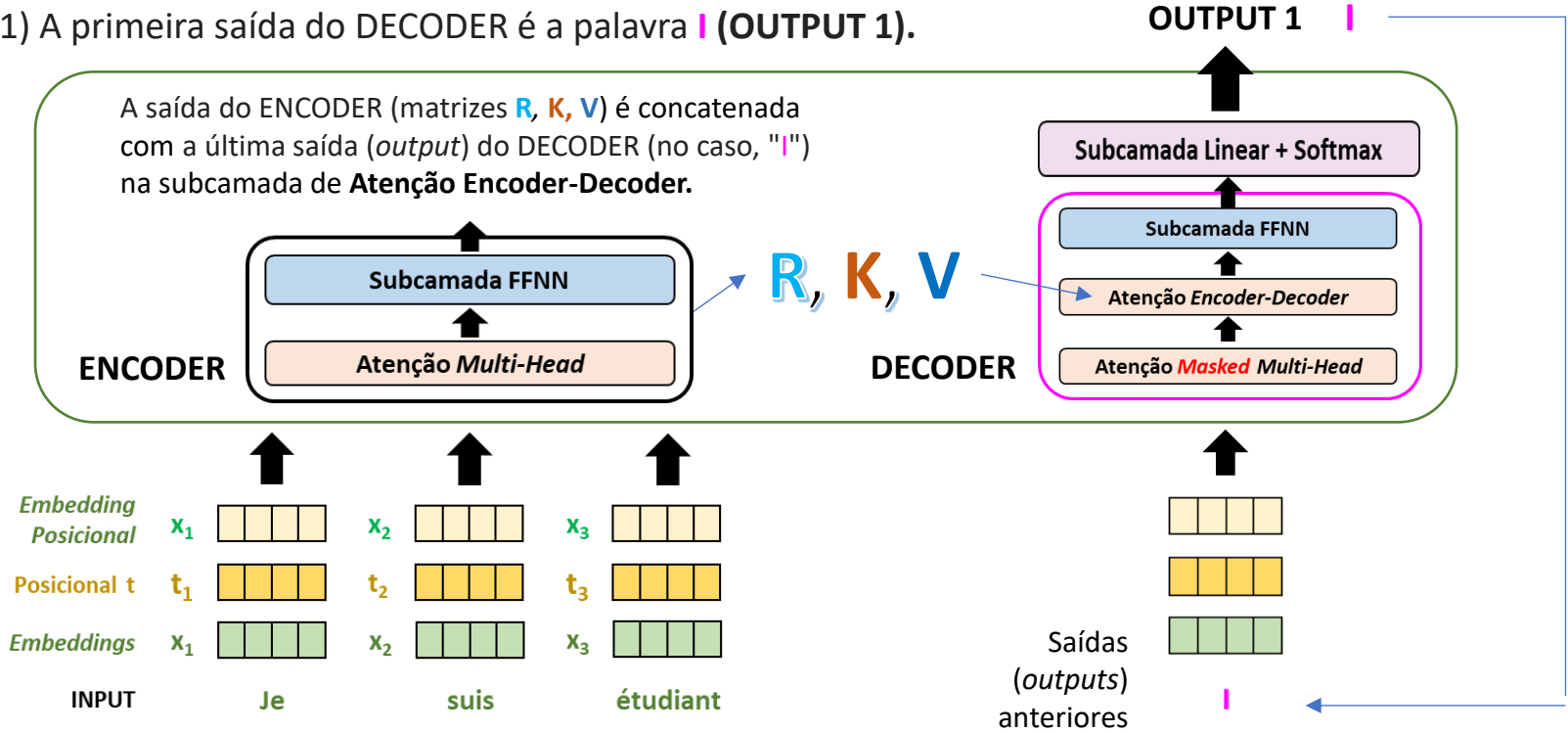
Ainda nos ENCODERS, a **representação final R** é usada para gerar a **matriz de chaves (K)** e a **matriz de valores (V)**. O processo envolve uma projeção linear da **representação final R** (são aplicadas duas transformações lineares diferentes para gerar as matrizes **K** e **V** mas podemos ignorar estes detalhes). Já no DECODER, a subcamada de **Atenção Encoder-Decoder** usa as **matrizes de chaves (K)** e de **valores (V)** para calcular a Atenção entre a sequência de entrada (contexto codificado) e a sequência de saída (decodificação). Isso permite que o modelo se concentre em diferentes partes da entrada durante a produção de cada palavra na sequência de saída.



4.5.2.1. Decodificação passo a passo

O processo de decodificação começa com um *token* de início de sequência como <sos> (*start of sequence*). Em cada passo na decodificação é gerada a palavra (*token*) de maior probabilidade para a sentença de saída (a tradução).

(1) A primeira saída do DECODER é a palavra **I** (OUTPUT 1).



**Feedback** - A primeira palavra gerada como saída ("**I**") pelos DECODERS (*output*) é realimentada na rede. É gerado um **vetor de Embeddings**, que é adicionado a um vetor *Posicional*, e o vetor resultante (*Embedding with Time Signal*) é realimentado na camada de **Atenção Mascarada** (*Masked Multi-Head Attention*) do DECODER. Assim, o modelo pode atentar para todas as posições *anteriores* na sequência durante a geração da palavra seguinte.

Figura 154: Fonte: Adaptada de [105] pelo autor

(2) A segunda saída do DECODER é a palavra **am** (OUTPUT 2).

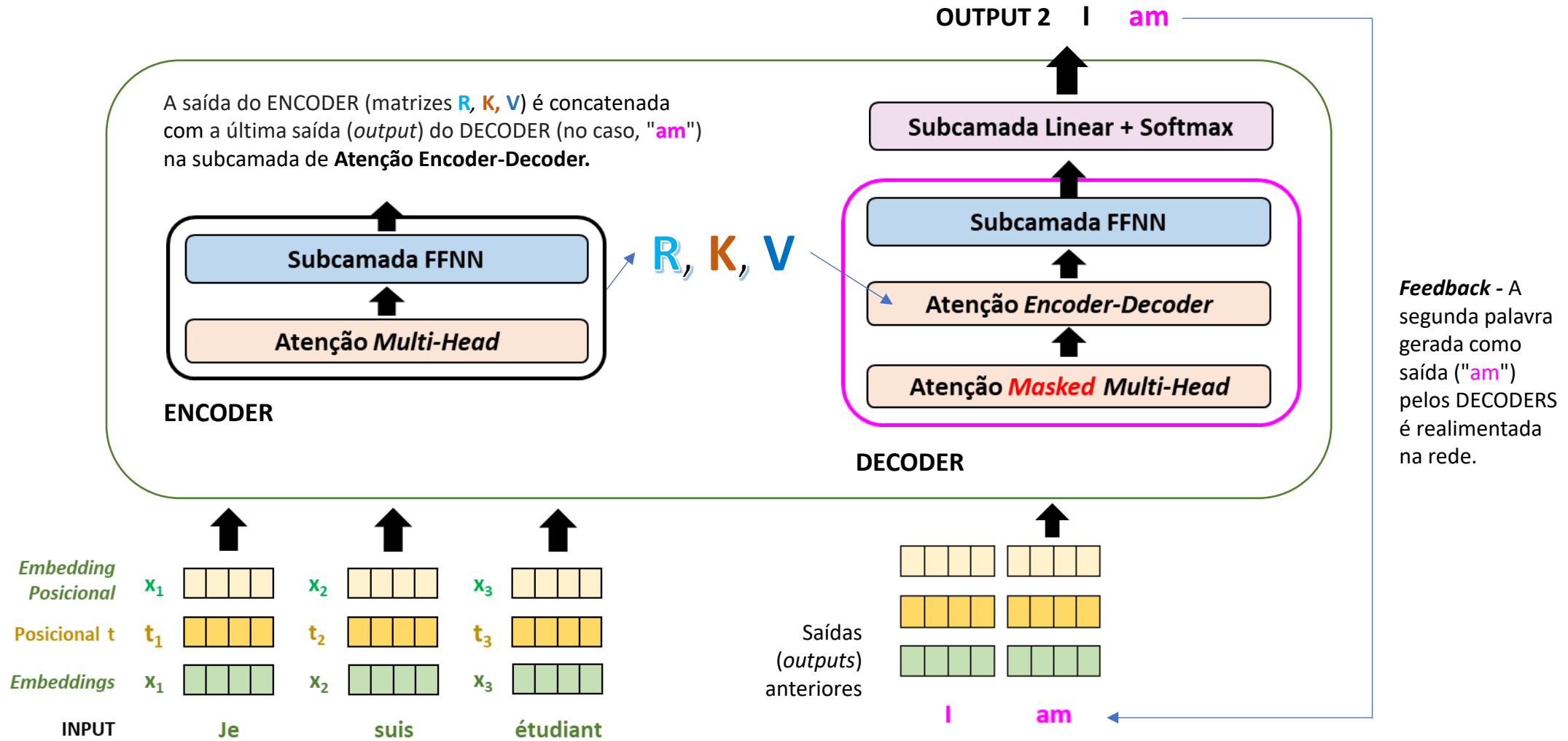


Figura 155: Fonte: Adaptada de [105] pelo autor

(3) A terceira saída do DECODER é a palavra **a** (OUTPUT 3).

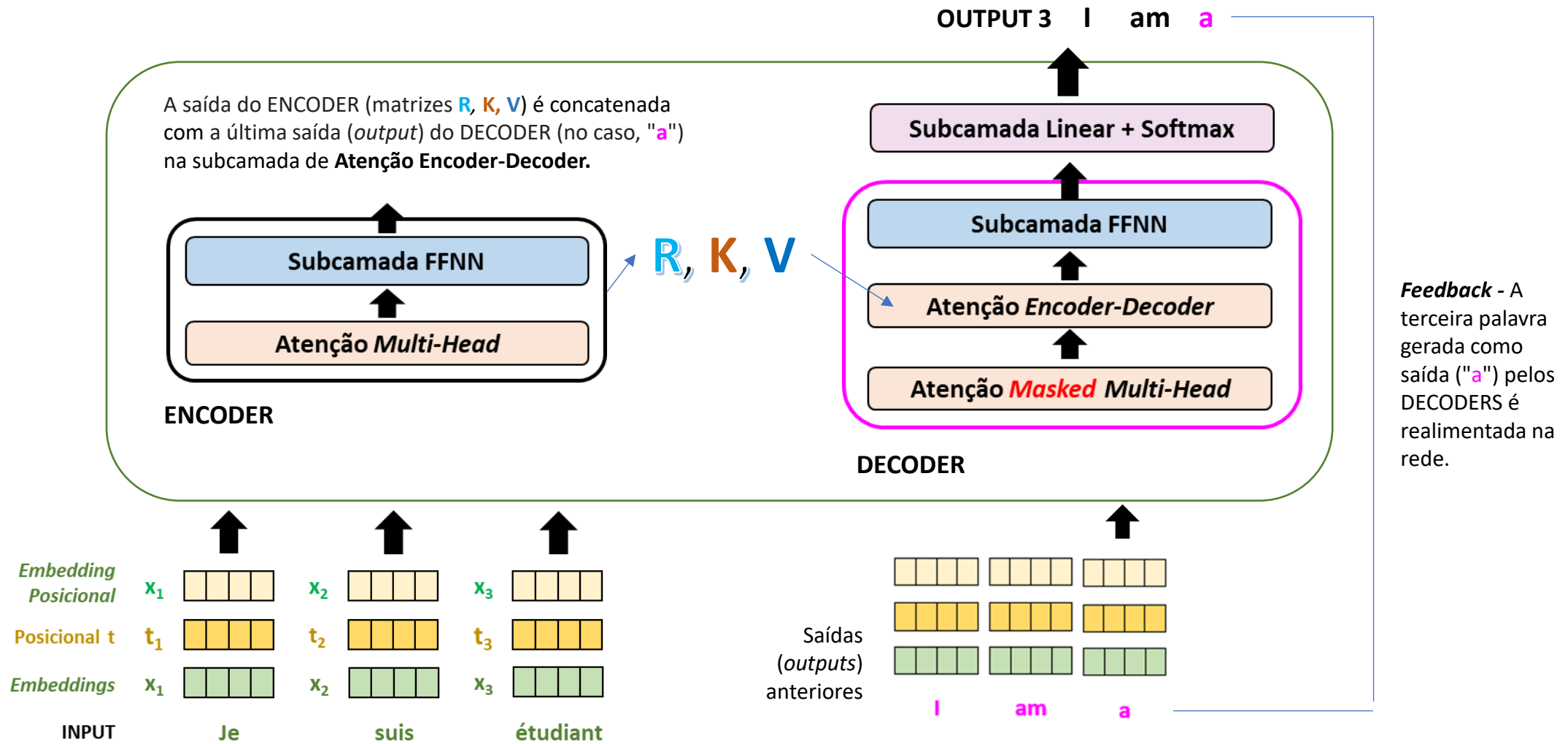


Figura 156: Fonte: Adaptada de [105] pelo autor

(4) A quarta é a palavra **student** (OUTPUT 4).

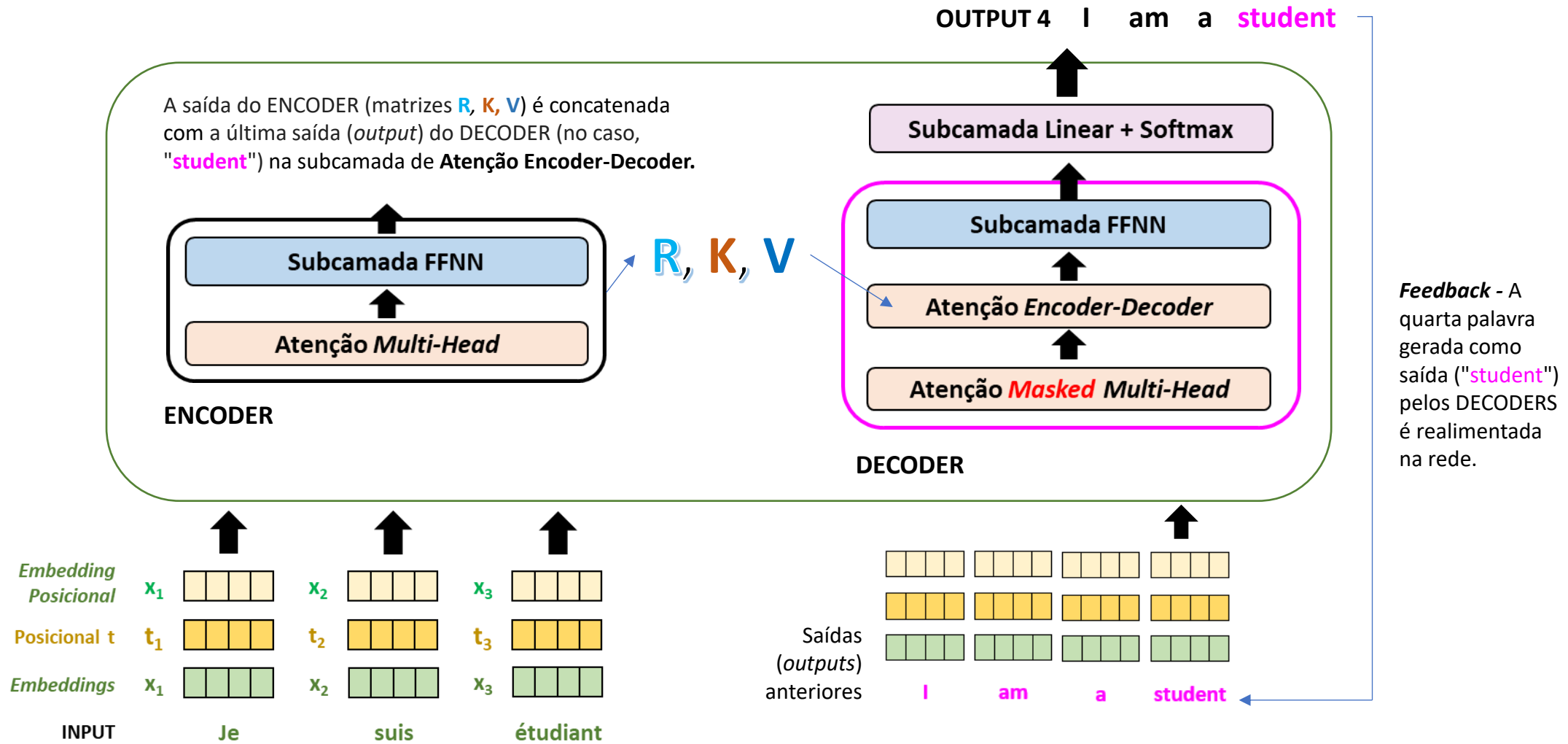


Figura 157: Fonte: Adaptada de [105] pelo autor

Um *token eos* <end of sequence> indica que o DECODER processou toda a sequência, e o processo é finalizado.

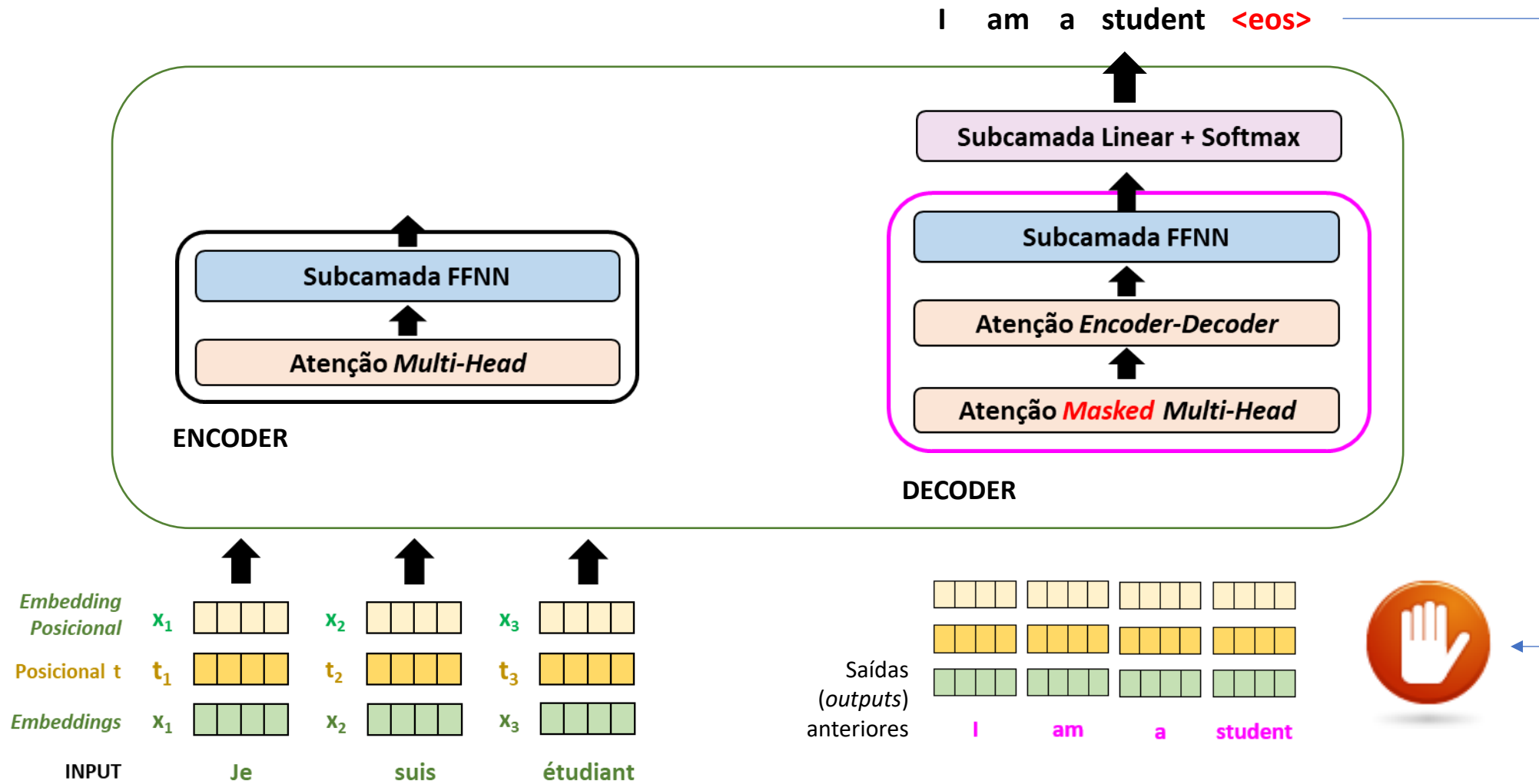


Figura 158: Fonte: Adaptada de [105] pelo autor

## Revisando:

### 1. Entrada (ENCODER)

A sentença de entrada, por exemplo, "jé suis étudiant", é tokenizada e incorporada em **vetores de Embeddings Posicionais**. Esses vetores são alimentados no primeiro codificador da pilha de ENCODERS do *Transformer*, e passam pelas subcamadas de Auto-Atenção e FFNN, gerando uma **representação final R**.

### 2. Saída (ENCODER)

A saída da última camada do último ENCODER é a **representação R final** da sequência de entrada, uma matriz que contém informações sobre *todas* as palavras na sequência de entrada e suas relações. A **representação final (R)** da sequência de entrada (obtida após a passagem por todas as camadas do ENCODER) é crucial para a fase de decodificação - cada elemento da **matriz R** contém informações sobre a respectiva palavra na sequência de entrada, capturando seu significado contextual em relação às outras palavras.

### 3. Vetores-Chave (K) e Vetores-Valor (V)

Ainda nos ENCODERS, a **representação R final** é usada para gerar a **matriz de vetores-chave (K)** e a **matriz de vetores-valor (V)**, como explicado no BOX 10. Essas matrizes serão usadas na subcamada de Atenção Encoder-Decoder de cada decodificador já na pilha de DECODERS, durante a geração da tradução.

### 5. Decodificação (DECODER)

O processo de decodificação começa com um *token* especial de início de sequência, como <sos> (*start of sequence*). O DECODER gera a próxima palavra mais provável para a tradução da sequência de entrada, por exemplo, "am", através da aplicação da função softmax nos *logits* produzidos pela última camada.

## 5. Realimentação (*Feedback*)

Vejamos o processo de realimentação que ocorre nos DECODERs, pois ele é importante na compreensão da arquitetura *Transformer*. É mais fácil acompanhar olhando a Figura 159 "de baixo para cima".

(a) Para cada palavra gerada como saída ("am" por exemplo) pelos DECODERS (*output*) é gerado um **veto**r de **Embeddings** (*Output Embedding*), que é somado ao vetor para Codificação Posicional (*Positional Encoding*), e o vetor resultante é realimentado na camada de Atenção Mascarada *Multi-Head* do DECODER. Assim, o modelo pode atentar para todas as posições *anteriores* na sequência durante a geração da palavra seguinte.

(b) A última saída (*output*) do DECODER (a última palavra traduzida no exemplo) é também concatenada na subcamada de Atenção Encoder-Decoder com a **representação final (R)**, os **vetores-chave (K)**, e os **vetores-valor (V)** gerados pelo bloco de ENCODERS.

Essa concatenação (Figura 159) permite ao modelo levar em conta tanto as palavras previamente geradas durante a tradução (saída), quanto o contexto da sequência de entrada gerado pelos ENCODERS, possibilitando uma Atenção mais abrangente, o que ajuda na tradução da *próxima* palavra da sequência.

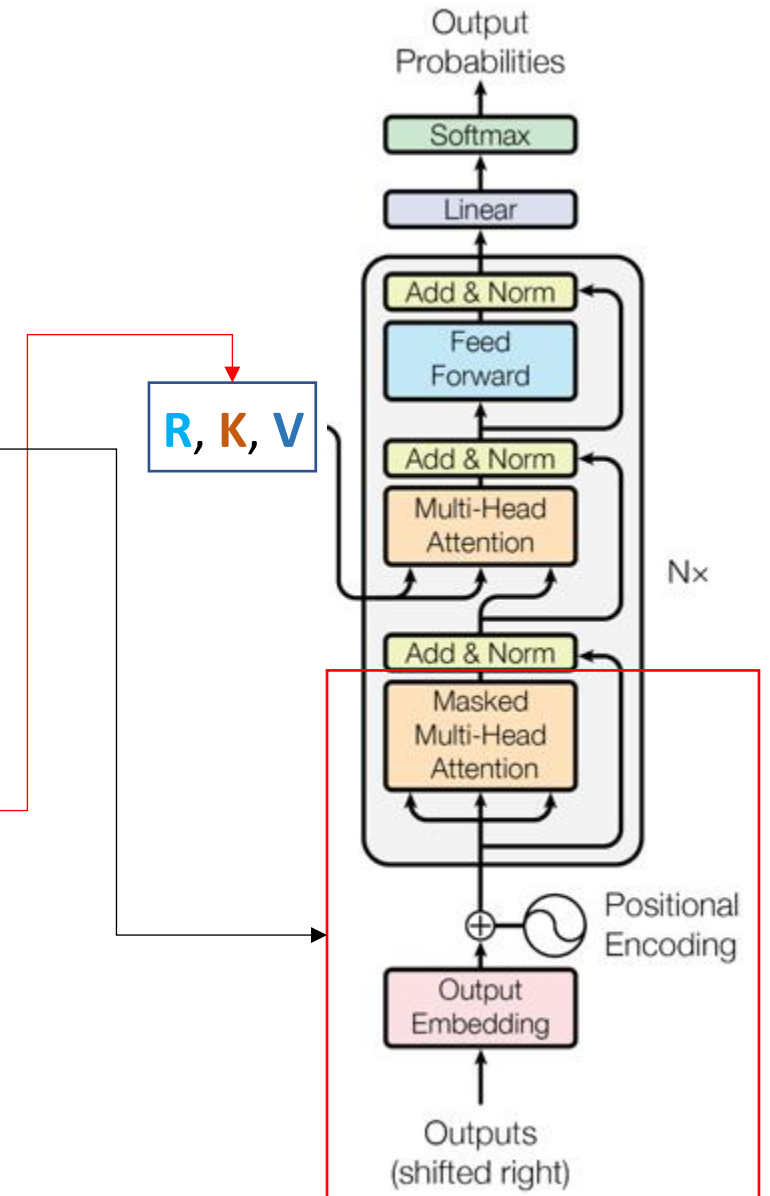


Figura 159: Fonte: Adaptado de [80]

## 6. Geração de Palavras Sucessivas

O processo de realimentação na subcamada de *Masked Multi-Head Attention* do DECODER ocorre *para cada saída (token)* gerado. Ou seja, se forem geradas quatro palavras como saída da tradução (I am a student), a realimentação vai ocorrer quatro vezes. Assim, cada palavra já traduzida é realimentada da rede para ajudar na tradução da seguinte, e o processo continua iterativamente. A realimentação melhora a qualidade e a coerência da tradução.

O processo de receber as matrizes **R**, **K** e **V** como saída final dos ENCODERS e passá-las para a subcamada de Atenção Encoder-Decoder nos DECODERS também ocorre diversas vezes. Na verdade, cada camada do DECODER recebe essas matrizes em cada passo de tempo durante o processo de geração de palavras. Isso é fundamental na arquitetura Transformer, pois permite que o modelo leve em consideração as *informações da sequência de entrada* em cada etapa da geração da sequência de saída (no caso, a tradução).

Em cada etapa, a subcamada de Atenção Encoder-Decoder nos DECODERS usa essas matrizes para realizar a Atenção *entre a palavra atual sendo gerada* (a palavra que está sendo traduzida) e as palavras da sequência de entrada (no exemplo, "Je suis étudiant"). Este processo vai se repetir para *cada camada* do DECODER e para *cada palavra* gerada durante a tradução. Considerando uma pilha com seis DECODERS e uma sequência com quatro palavras, este processo vai se repetir 24 vezes, até que a sequência de saída seja completamente gerada.

O modelo continua gerando palavras até que um *token* de fim de sequência, como <eos> (*end of sequence*), seja gerado ou até que seja atingido um comprimento máximo predefinido para a sequência de tradução (por exemplo, um determinado limite de *tokens* de saída).



#### 4.5.2.2. Soma de saídas e Normalização

Tal como ocorre nos ENCODERS, nos DECODERS também é utilizada a camada de soma e normalização (**Add & Norm**), em amarelo.

Como vimos (BOX 9), na camada a saída do mecanismo de Atenção (um vetor de números reais) passa por duas operações, uma adição (**Add**) seguida de uma normalização (**Norm**). Esse processo é repetido para cada subcamada do DECODER.

- **Add:** A saída da subcamada de Auto-Atenção é somada à entrada original (é uma soma de matrizes). Esta forma de conexão residual permite que as informações originais da entrada sejam preservadas, mitigando problemas dos "Gradientes que Desaparecem" (na falta de uma tradução melhor para *Vanishing Gradients*):
- **Norm:** Após a adição, a saída é submetida à normalização por camada. Isso implica aplicar uma transformação linear que normaliza a média e o desvio padrão da saída.

Como destacado na Figura 160, a saída da subcamada **Add & Norm** é utilizada como entrada para a subcamada FFNN (*Feed Forward*).

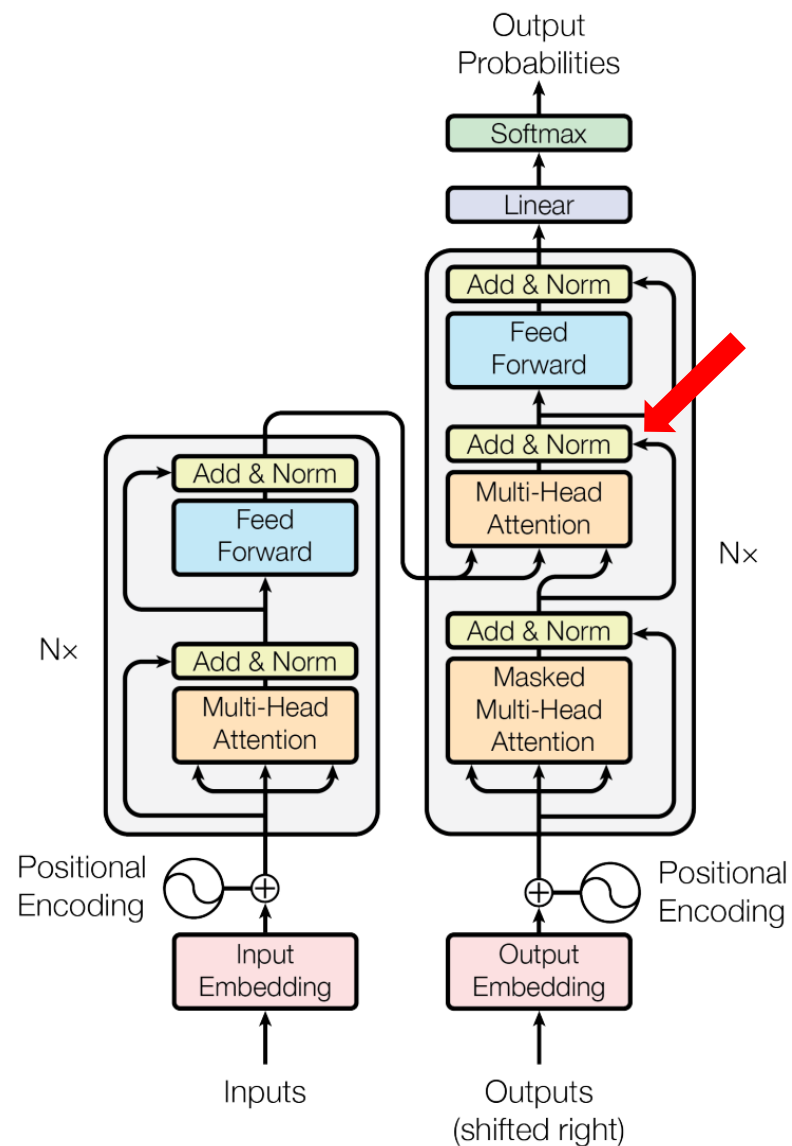


Figura 160: Fonte: [80]

#### 4.5.2.3. A subcamada FFNN nos DECODERS

Na subcamada FFNN (*Feed Forward Neural Network*) de cada DECODER o vetor resultante das camadas de Atenção (já devidamente normalizado) passa por transformações para refinar ainda mais a representação contextualizada.

Mais precisamente, cada posição da sequência (cada *token*) passa por uma *Rede Neural Feed Forward*. Esta rede geralmente consiste em duas operações lineares (transformações afins) com uma função de ativação não linear aplicada entre elas. Aqui, é utilizada a função de ativação ReLU (*Rectified Linear Unit*) [101].

A saída da subcamada FFNN é uma nova representação vetorial para cada posição na sequência. Essa representação captura padrões complexos e interações não lineares na informação da entrada (isso significa que "a compreensão de certas relações complexas entre as palavras que ocorrem na linguagem natural melhora").

Como podemos ver na Figura 161, após passar pela subcamada FFNN de cada DECODER, há uma nova operação de adição e normalização, e em seguida o resultado é entregue para a camada **Linear** e a camada **Softmax** como veremos a seguir.

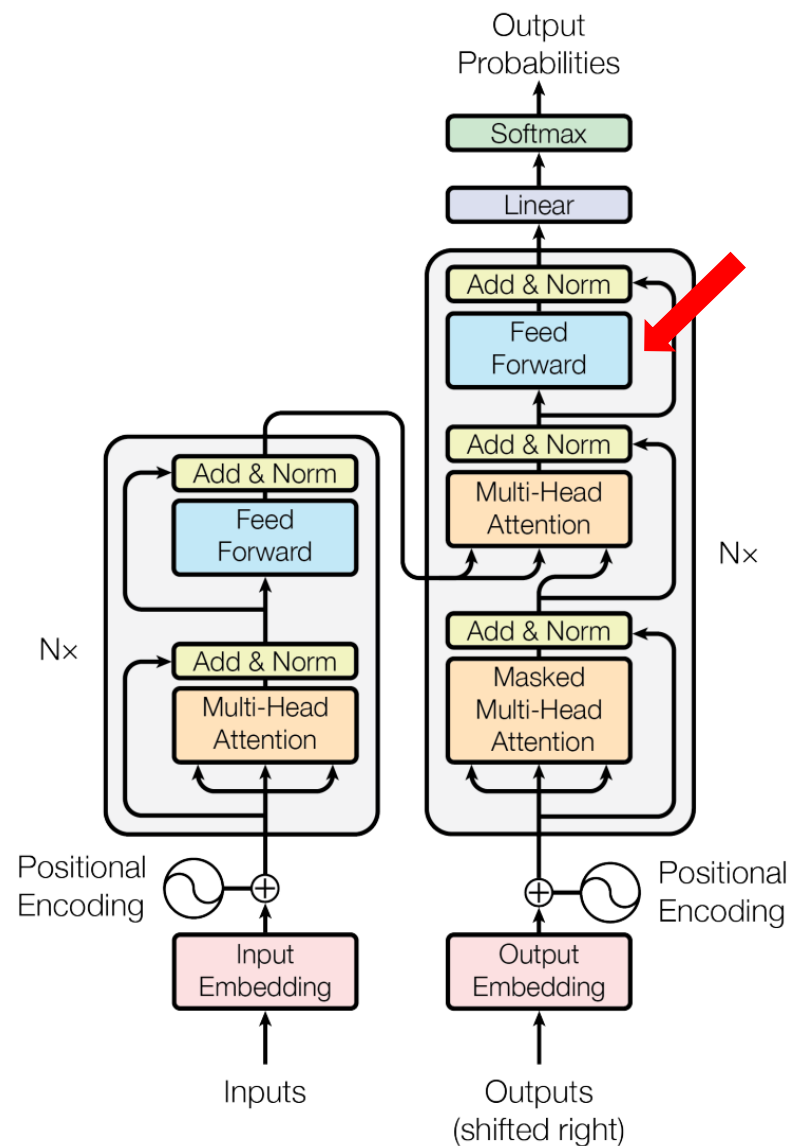


Figura 161: Fonte: [80]

#### 4.5.2.4. Camada Linear e camada Softmax

Cada saída do DECODER é um *token*, cuja representação interna para as redes neurais (que "gostam de números") é um vetor numérico. O problema é que ao contrário das redes neurais os humanos preferem ler textos do que vetores de números reais, portanto é chegada a hora de transformar as saídas em algo mais inteligível - palavras da linguagem natural.

A camada **Linear** (em destaque, Figura 162) é uma outra rede neural (às vezes chamada de *camada totalmente conectada*) na arquitetura *Transformer* que faz uma projeção (operação matemática) do vetor de saída com uma representação latente gerada pela pilha de DECODERS em um vetor muito maior, que é o vetor de *logits*, já discutido na Seção 2.8 e também no BOX3.

Relembrando: O vetor de *logits* contém valores numéricos que representam as saídas brutas do modelo *antes* da aplicação da função softmax. Cada elemento do vetor é um número real (positivo, nulo ou negativo) *atribuído a cada token no vocabulário* (podem ter, por exemplo, 50.000 elementos). Os *logits ainda não são probabilidades*, eles indicam a maior ou menor "preferência" do modelo por determinada categoria ou classe para o *token* que está sendo gerado.

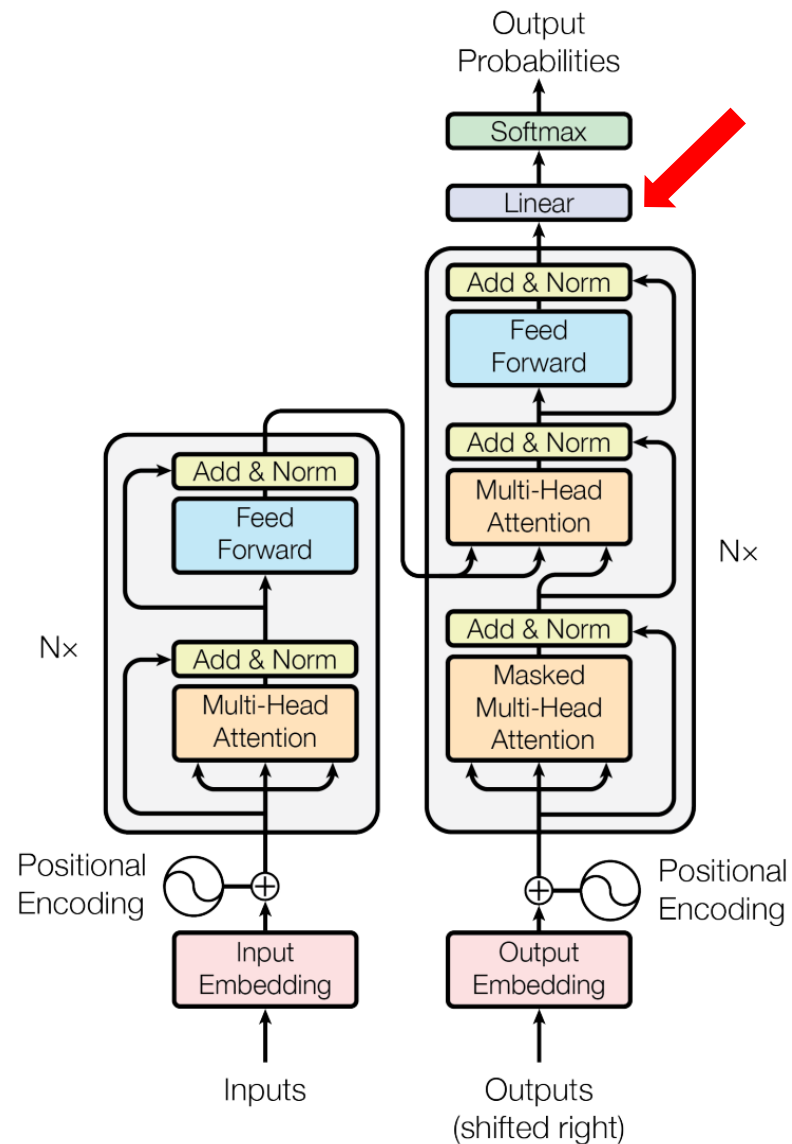


Figura 162: Fonte: [80]

Feita a projeção do vetor de saída gerado pelos DECODERS no vetor de *logits*, a camada **softmax** transforma os valores numéricos com as "preferências" ou pontuações do modelo em probabilidades, gerando um novo vetor (vetor de probabilidades) normalizado, cujos elementos são todos positivos, e a soma de todos os valores é igual a 1 ou 100%.

O elemento com maior probabilidade é escolhido (**argmax**), ou pode-se utilizar amostragem com "temperatura" para permitir que outros *tokens* um pouco menos prováveis sejam escolhidos (dependendo da tarefa). Este elemento ainda é um *token*, mas sabemos que os *tokens* são indexados, eles têm um índice numérico que permite mapeá-los para palavras do vocabulário. Então, como saída final é feita a conversão do *token* mais provável para uma palavra em linguagem natural. Este processo se repete em cada passo, ou seja, a cada palavra gerada pela pilha de DECODERS.

OUTPUT 4    I    am    a ...

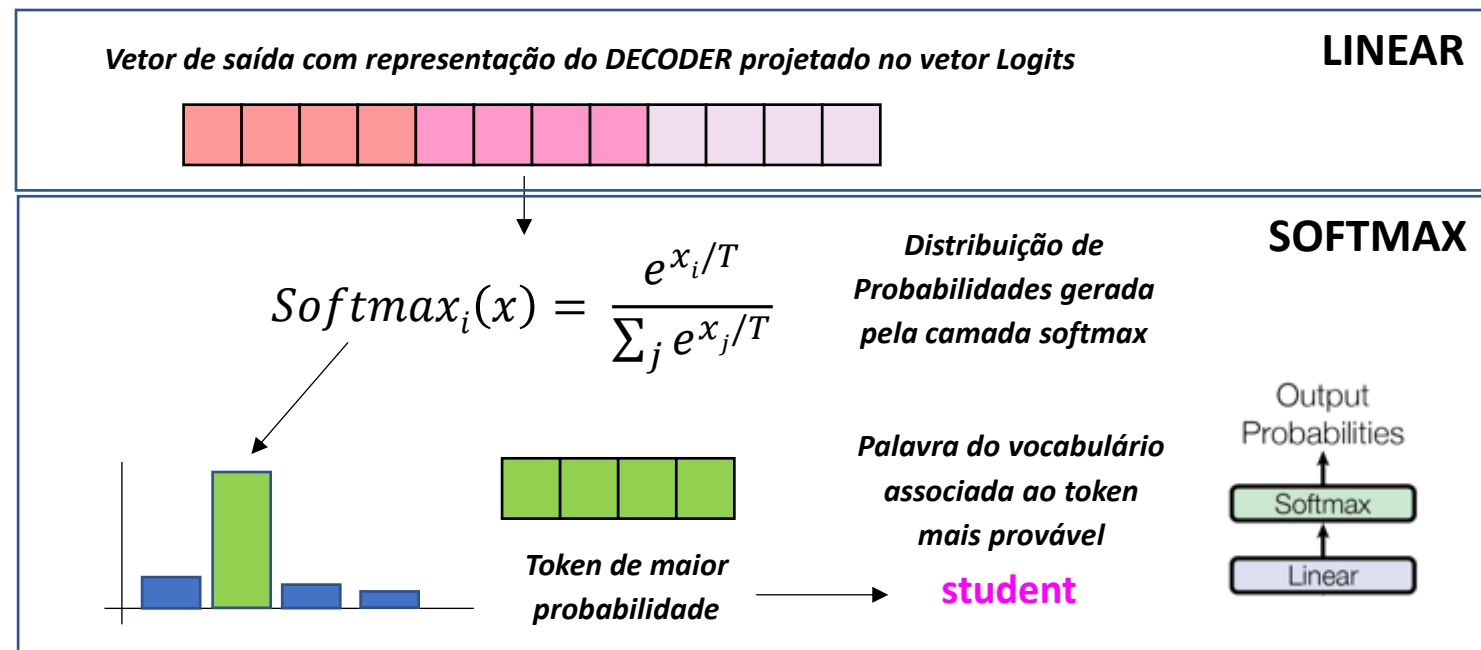


Figura 163: Fonte: Adaptada de [105] pelo autor

## 4.6. Revisão

Em uma entrevista de 8 minutos [110] com Lex Fridman, Andrej Karpathy explica em termos simples como funciona a arquitetura *Transformer*. Vamos utilizar sua explicação como uma revisão do que vimos até agora.

- O *Transformer* é uma magnífica arquitetura de redes neurais, que atua como um "computador de propósito geral". Pode ser utilizada para resolver diferentes tipos de problemas em IA, tanto de NLP (processamento de linguagem natural) quanto de visão computacional.
- A rede tem uma série de blocos contendo nós (neurônios) que armazenam *vetores*. Graças ao mecanismo de Atenção, estes nós podem "observar uns aos outros", ou seja, cada nó pode consultar os vetores armazenados em outros nós, e os nós podem se comunicar. Um nó pode fazer um *broadcast* tipo "ei, estou procurando tal informação", outros nós podem responder "estas são as informações que eu tenho (vetores e seus valores)", e assim os nós podem se atualizar progressivamente, melhorando a performance do modelo.
- É uma arquitetura **expressiva** (= capaz de expressar diversos tipos de algoritmos), composta por muitos componentes interessantes além da Atenção, como as normalizações, a função softmax e as *conexões residuais*.
- É **otimizável** (através de ajuste de pesos), o que é importante pois "há muitos computadores que são altamente poderosos, mas não podem ser facilmente otimizados através das técnicas atualmente disponíveis". No caso, o *Transformer* pode ser otimizado por *Backpropagation + Gradient Descent*.
- É **eficiente** (alto paralelismo computacional) e adequada para o melhor hardware disponível atualmente (GPUs). Estes supercomputadores virtuais compostos por milhares de GPUs utilizados no treinamento de LLMs por exemplo são projetados para o processamento em paralelo (em oposição à operações sequenciais), e a arquitetura *Transformer* foi projetada com isso em mente.

Apesar da simplicidade nas colocações de Andrej Karpathy, não se deixe enganar imaginando que o processo que ocorre no treinamento de um grande modelo de linguagem com a arquitetura *Transformer* (ou, a bem da verdade, em qualquer outro processo que envolva *Deep Learning* e redes neurais complexas) é trivial ou simples. Pelo contrário, há muitos detalhes técnicos que não foram mencionados aqui, muitos truques e ajustes precisam ser feitos para que tudo funcione. Na verdade, *é bem difícil fazer tudo corretamente*. Por exemplo, durante o treinamento é preciso utilizar um otimizador ADAM (*Adaptive Moment Estimations*) [111] para redução do gradiente (minimizar os erros da rede). Para uma otimização mais estável e para reduzir o risco de *overfitting*, alguns *dropouts* precisam ser executados em cada camada [112]. Também é preciso fazer *label smoothing* [113], uma técnica de regularização para melhorar a calibragem nas previsões probabilísticas do modelo (um modelo de classificação está calibrado quando suas previsões de probabilidades são compatíveis com a sua acurácia). Assim, na prática, o treinamento de LLMs com redes neurais profundas é um assunto para especialistas, não só pelas dificuldades técnicas mas também pela enorme infraestrutura de computação demandada. Porém, esperamos que o que discutimos já traga um entendimento razoável do assunto.

A arquitetura *Transformer* tem se mostrado bastante **resiliente e estável**, e ainda não foi superada (desde 2016) por qualquer outra, e já está inclusive sendo adaptada para outras tarefas além do processamento de linguagem. No *Podcast* com Fridman [110], ao ser perguntado sobre "qual era a ideia mais bela e surpreendente que havia surgido no campo do *Deep Learning* ou mesmo da Inteligência Artificial de modo geral", Andrej Karpathy respondeu "a arquitetura *Transformer*, que permitiu a convergência de vários ramos da IA que eram até então atendidos por diferentes tipos de redes neurais (como visão computacional, processamento de linguagem e outros) para uma única arquitetura", e completou "podemos passar como entradas dados de diferentes tipos, como textos, imagens, vídeos, áudios etc. e treinar vários modelos de IA com esta mesma arquitetura, feitos alguns ajustes". Ou seja, além de mais eficiente em termos computacionais a arquitetura *Transformer* é "suficientemente flexível para resolver diferentes tipos de problemas, desde prever a próxima palavra em uma frase ou detectar se há um gato em uma imagem".

# Conclusão e referências

## Conclusão

Pesquisas recentes [114] têm revelado que os grandes modelos de linguagem podem produzir muitas capacidades ou **comportamentos emergentes** que não eram previstos, e várias destas capacidades têm pouca relação com apenas analisar ou completar textos, como fazer operações aritméticas e gerar códigos de computador. A princípio, os modelos deveriam "receber uma sequência de texto e prever a próxima palavra mais provável para completá-lo", mas ninguém realmente imaginava que estes modelos fossem demonstrar tantos novos "talentos" - como adivinhar o nome do filme com base apenas em alguns emojis (Figura 164).

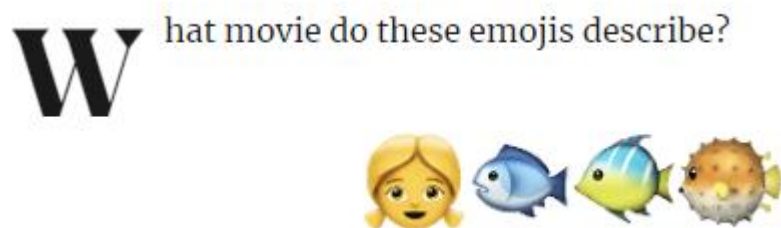


Figura 164: Fonte: [114]

Figura 164 - Um *Prompt* composto por quatro emojis e a pergunta "Que filme é descrito por estes emojis?" foi uma das 204 tarefas planejadas para testar as habilidades de diversos grandes modelos de linguagem que impulsionam o ChatGPT e outros assistentes. Os modelos mais simples geraram respostas sem qualquer sentido como "O filme é sobre um homem que é um homem que é um homem". Modelos de média complexidade (em termos de quantidade de parâmetros) chegaram mais perto, "chutando" como resposta "O filme do Emoji". Porém, o modelo mais complexo acertou a resposta em cheio: "Procurando Nemo" [114].

Os pesquisadores verificaram que estas propriedades emergentes não surgiam apenas como resultado da "complexidade" ou com quantos bilhões de parâmetros o modelo foi treinado. Algumas habilidades inesperadas foram reveladas mesmo por modelos menores e treinados com menos dados, desde que os dados tivessem alta qualidade.

Um outro fator relevante na acurácia das respostas é a forma como as instruções (*Prompts*) são construídas. Como vimos na Seção 3.3, "se você quer um modelo mais preciso, peça por isso", ou seja, o uso de *Chain of Thought Reasoning* [89] é capaz de produzir respostas mais acuradas em tarefas envolvendo aritmética ou raciocínios lógicos, e o fato de se obter mais acurácia apenas por pedir ao modelo para prosseguir passo a passo e explicar sua própria resposta já é uma propriedade emergente dos LLMs que não havia sido prevista.

Como vimos, grande parte do sucesso do ChatGPT e outras aplicações baseadas em LLMs pode ser creditada para a arquitetura *Transformer*. Como mencionado por Łukasz Kaiser em sua Masterclass [103], "o que a rede neural *Transformer* faz até pouco tempo os computadores não eram capazes de fazer", e em grande parte isso se deve aos mecanismos de Atenção.

Escrevemos "mecanismos" no plural, pois como vimos há três tipos de Atenção diferentes na arquitetura:

- A Auto-Atenção nos ENCODERS (*Multi-Head Attention*), que é uma atenção da entrada em si própria (por isso é chamada de Auto-Atenção ou *Self Attention*).
- A Atenção entre os ENCODERS e os DECODERS (*Encoder-Decoder Attention*), onde a saída (DECODER) presta atenção na entrada (representação passada pelo ENCODER).
- A Atenção mascarada nos DECODERS (*Masked Multi-Head Attention*), uma matriz multiplicada por uma máscara, onde a saída presta atenção em saídas geradas anteriormente.



Já que foi tão bem sucedida treinando grandes modelos para resolver tarefas associadas com a linguagem natural (tradução, analisar sentimentos, completar textos etc.), é normal que os pesquisadores investiguem para que mais a arquitetura *Transformer* pode ser utilizada.

Ainda não há uma resposta definitiva para esta pergunta, mas vários testes estão sendo realizados, e até agora a arquitetura está se mostrando surpreendentemente versátil, tal como mencionado por Andrej Karpathy.

Um exemplo é a adaptação da arquitetura *Transformer* para treinar modelos de classificação e outros em Visão Computacional para executar tarefas como processar e classificar imagens. Sobre isso, merece atenção o trabalho que está sendo desenvolvido pelo pesquisador Alexey Dosovitskiy da Google [115, 116], que com seus colegas está desenvolvendo a **Vision Transformer (ViT)**, uma adaptação da arquitetura *Transformer* para Visão Computacional, e já obteve resultados excelentes em comparação com as Redes Neurais Convolucionais (CNN - *Convolutional Neural Network*), que até então eram o estado da arte para este tipo de tarefa.

Além disso, a boa performance foi obtida utilizando substancialmente menos recursos computacionais durante o treinamento (o que reduz o prazo e os custos dos projetos, dentre outros benefícios). A rede neural **Vision Transformer (ViT)** consegue classificar imagens com acurácia de 90% - um resultado bem superior ao que os especialistas esperavam, o que colocou a **ViT** já no topo de um *ranking* de um famoso desafio anual entre algoritmos de reconhecimento de imagens (*The ImageNet Large Scale Visual Recognition Challenge*) [117].

Ao que parece, os *Transformers* (com pequenas adaptações em relação à arquitetura original) têm potencial viabilizar uma inédita e benéfica convergência entre dois importantíssimos campos do *Machine Learning*, o NLP (Processamento de Linguagem Natural) e a Visão Computacional, que até poucos anos atrás evoluíam em trilhas separadas, cada um com suas técnicas, algoritmos e métodos.

Como não há lanches grátis, é pertinente reforçar os alertas já feitos por pesquisadores da Anthropic [118] de que as capacidades emergentes e inesperadas dos LLMs trazem imprevisibilidade para o bem ou para o mal - podem surgir benefícios inesperados, mas também podem surgir riscos não previstos, como toxicidade nas respostas ou novos tipos de *bias* raciais ou sociais. Por outro lado, estes mesmos comportamentos emergentes imprevisíveis dos grandes modelos talvez possa ajudar a *mitigar* alguns destes riscos! Por exemplo, quando os pesquisadores da Anthropic instruíram alguns modelos *explicitamente* para não se basearem em estereótipos ou preconceitos sociais, e para além disso utilizaram CoT (*Chain of Thought Prompting*), as predições geradas mostraram *menos* bias (Figura 165). Isso sugere que talvez os grandes modelos tenham capacidade de "auto-correção moral" [119] e possam mitigar o seu próprio *bias*.

Figura 165 - Quanto maior o modelo (bilhões de parâmetros), maior o risco de *bias*. A linha marrom mostra que na medida em que os modelos se tornam maiores (eixo x) o *bias* aumenta (eixo y). Por outro lado, na medida em que os modelos se tornam maiores, eles se tornam cada vez mais capazes de reduzir o *bias* que eles próprios geram, se forem adequadamente instruídos para fazê-lo [119].

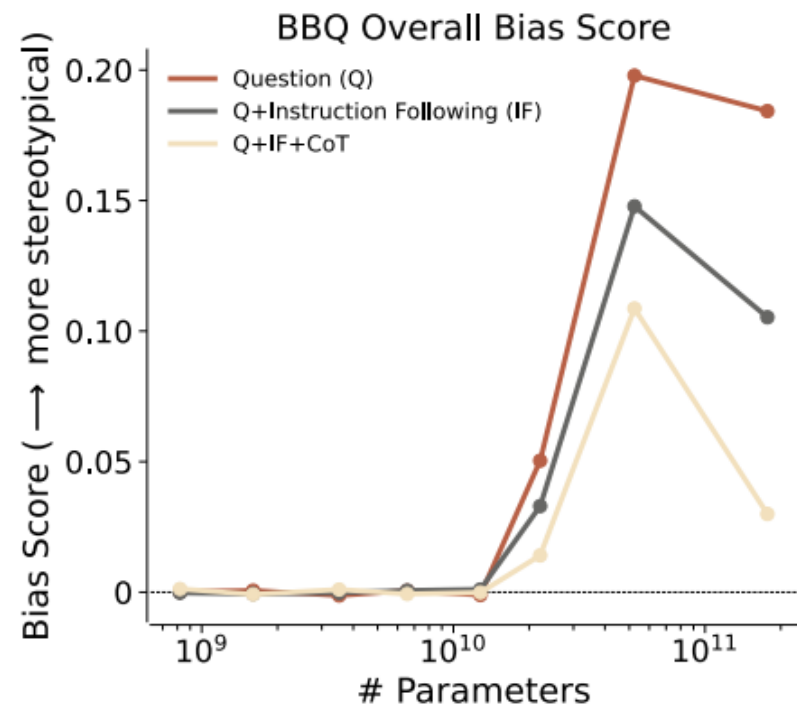


Figura 165: Fonte: [119]

Isso sinaliza que há um *vasto campo ainda a ser pesquisado*, e de grande relevância - como utilizar comportamentos emergentes dos LLMs para reduzir riscos causados pelos comportamento emergentes dos LLMs?

Naturalmente, não podemos contar apenas que os grandes modelos de linguagem "fiscalizem a si próprios com sua auto-correção moral", e o mesmo pode ser dito sobre qualquer *Big Tech* poderosa o suficiente para criá-los. Assim, é importante encontrar rapidamente soluções adequadas para o grande *gap* regulatório existente para controle das aplicações de Inteligência Artificial que já estão no mercado (*e outras ainda mais poderosas que virão*), pois como sabemos os grandes modelos de IA Generativa evoluem muito rapidamente, e ainda não há padrões ou normativas legais formalmente estabelecidas para regulamentar a sua utilização, embora existam esforços em andamento em todo o mundo, inclusive no Brasil [120].

Concluindo, esperamos que as informações aqui compartilhadas tenham sido suficientes para dar ao menos uma noção básica de "como funciona o Assistente ChatGPT", pergunta que nos levou a pesquisar como os grandes modelos de linguagem são treinados, o que por sua vez nos conduziu para as redes *Transformer* e seus recursos de Atenção.

O autor não é especialista em nenhum dos assuntos aqui discutidos, e se desculpa por qualquer imprecisão ou omissão importante. Caso alguma parte do texto não tenha ficado suficientemente clara, consulte as referências sugeridas na Seção seguinte, e assim você poderá tirar suas dúvidas diretamente de fontes mais técnicas.

*Atenção é tudo o que você precisa.*

# Referências

1. *ChatGPT*

<https://chat.openai.com/>

2. *OpenAI*

<https://openai.com/>

3. *GPT-3*

<https://platform.openai.com/docs/models/gpt-3>

4. *GPT-3.5*

<https://platform.openai.com/docs/models/gpt-3-5>

5. *GPT-4*

<https://openai.com/gpt-4>

6. *OpenAI GPT Models - Lei Mao*

<https://leimao.github.io/article/OpenAI-GPT-Models/>

7. *ChatGPT and OpenAI's use of Azure's Cloud Infrastructure By Mary Zhang -January 26, 2023*

<https://dgtlinfra.com/chatgpt-openai-azure-cloud/>

8. *ChatGPT*

<https://chat.openai.com/auth/login>

9. *Is ChatGPT biased?*

<https://help.openai.com/en/articles/8313359-is-chatgpt-biased>

10. *Google Bard*

<https://bard.google.com/chat>

11. *Vicuna*

<https://lmsys.org/blog/2023-03-30-vicuna/>

12. *GPUs NVIDIA*

<https://developer.nvidia.com/deep-learning>

13. *OpenAI Codex*

<https://openai.com/blog/openai-codex>

14. *OpenAI DALL.E2*

<https://openai.com/dall-e-2>

15. *Language Models are Few-Shot Learners - OpenAI*

<https://arxiv.org/pdf/2005.14165.pdf>

16. *MetaAI LLaMA (Large Language Model Meta AI)*

<https://en.m.wikipedia.org/wiki/LLaMA>

17. *Natural language processing*

[https://en.wikipedia.org/wiki/Natural\\_language\\_processing](https://en.wikipedia.org/wiki/Natural_language_processing)

18. *ELMo*

<https://en.m.wikipedia.org/wiki/ELMo>

19. *BERT*

[https://en.m.wikipedia.org/wiki/BERT\\_\(language\\_model\)](https://en.m.wikipedia.org/wiki/BERT_(language_model))

20. *GPT-2: 1.5B release*

<https://openai.com/research/gpt-2-1-5b-release>

21. *Google LaMDA*

<https://blog.google/technology/ai/lamda/>

22. *Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model*

<https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/>

23. *PaLM (Pathways Language Model) from Google*

<https://en.m.wikipedia.org/wiki/PaLM>

24. *The Promise and Perils of Large Language Models*

<https://twosigmaventures.com/blog/article/the-promise-and-perils-of-large-language-models/>

25. *Gemini*

<https://deepmind.google/technologies/gemini/#introduction>

26. *Measuring Massive Multitask Language Understanding*

<https://arxiv.org/abs/2009.03300>

27. *From Syntax to Superpowers: The Marvelous Journey of NLP's Metamorphosis*

<https://blog.dataiku.com/nlp-metamorphosis?hsCtaTracking=35eb5e4b-9e0f-4c25-ac20-ef2d863005b7%7C4ff62fec-9c34-4e8c-8cb6-26a878216c65>

28. *Meta: LLaMA Language Model Outperforms OpenAI's GPT-3*

<https://aibusiness.com/meta/meta-s-llama-language-model-outperforms-openai-s-gpt-3>

29. *How to train your GPT Assistant, Andrej Karpathy*

<https://youtu.be/bZQun8Y4L2A?feature=shared>

30. *BioBERT: a pre-trained biomedical language representation model for biomedical text mining*

<https://arxiv.org/abs/1901.08746>

31. *LEGAL-BERT: The Muppets straight out of Law School*

<https://arxiv.org/abs/2010.02559>

32. *CamemBERT: a Tasty French Language Model*

<https://arxiv.org/abs/1911.03894>

33. *SciBERT: A Pretrained Language Model for Scientific Text*

<https://arxiv.org/abs/1903.10676>

34. *Introducing BloombergGPT, Bloomberg's 50-billion parameter large language model, purpose-built from scratch for finance*

<https://www.bloomberg.com/company/press/bloomberggpt-50-billion-parameter-llm-tuned-finance/>

35. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*  
[https://scontent.fsdu5-1.fna.fbcdn.net/v/t39.8562-6/106373513\\_3414102562251474\\_8005430471454563564\\_n.pdf?\\_nc\\_cat=105&ccb=1-7&\\_nc\\_sid=b8d81d&\\_nc\\_ohc=h8Pb-g2kgVoAX9oOJXv&\\_nc\\_ht=scontent.fsdu5-1.fna&oh=00\\_AfDxfeeNRnFCV4EHdTABrg1eU-HNR46SvQQ2\\_tQSocyKUA&oe=65579284](https://scontent.fsdu5-1.fna.fbcdn.net/v/t39.8562-6/106373513_3414102562251474_8005430471454563564_n.pdf?_nc_cat=105&ccb=1-7&_nc_sid=b8d81d&_nc_ohc=h8Pb-g2kgVoAX9oOJXv&_nc_ht=scontent.fsdu5-1.fna&oh=00_AfDxfeeNRnFCV4EHdTABrg1eU-HNR46SvQQ2_tQSocyKUA&oe=65579284)
36. *ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations*  
<https://blog.research.google/2019/12/albert-lite-bert-for-self-supervised.html>
37. *The SAT Reading Test*  
<https://satsuite.collegeboard.org/sat/whats-on-the-test/Reading>
38. *RACE: Large-scale ReAding Comprehension Dataset From Examinations*  
<https://aclanthology.org/D17-1082/>
39. *CLIP: Connecting text and images*  
<https://openai.com/research/clip>
40. *Stable Diffusion Online*  
<https://stablediffusionweb.com/>
41. *ProGen: Language Modeling for Protein Generation*  
<https://arxiv.org/abs/2004.03497>
42. *A Compact Guide to Large Language Models - databricks eBook.*  
<https://www.databricks.com/resources/ebook/tap-full-potential-llm>



43. *What Is ChatGPT Doing ... and Why Does It Work?* Stephen Wolfram

<https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/>

44. *Capítulo 15 Modelos de Linguagem.* Aline Paes et al.

<https://brasileiraspln.com/livro-pln/1a-edicao/parte7/cap15/cap15.html>

45. *ChatGPT is now available in Azure OpenAI Service*

<https://azure.microsoft.com/en-us/blog/chatgpt-is-now-available-in-azure-openai-service/>

46. *How to Use Large Language Models in the Enterprise - dataiku*

<https://blog.dataiku.com/llms-in-the-enterprise>

47. *Consultants Emerge as Early Winners in Generative AI Boom*

<https://www.wsj.com/articles/consultants-emerge-as-early-winners-in-generative-ai-boom-8df71d38>

48. *A new era of generative AI for everyone*

<https://www.accenture.com/us-en/services/applied-intelligence/generative-ai>

49. *The future of consulting in the age of Generative AI*

[https://www.ey.com/en\\_in/consulting/the-future-of-consulting-in-the-age-of-generative-ai](https://www.ey.com/en_in/consulting/the-future-of-consulting-in-the-age-of-generative-ai)

50. *The implications of Generative AI for businesses - A new frontier in Artificial Intelligence*

<https://www2.deloitte.com/us/en/pages/consulting/articles/generative-artificial-intelligence.html>

51. *O que é o Serviço OpenAI do Azure?*

<https://learn.microsoft.com/pt-br/azure/ai-services/openai/overview>

52. *Serviço OpenAI do Azure*

<https://azure.microsoft.com/pt-br/products/ai-services/openai-service>

53. *Customize a model with fine-tuning (preview)*

<https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/fine-tuning?tabs=turbo&pivots=programming-language-studio>

54. *The great acceleration: CIO perspectives on generative AI. MIT Technology Review & databricks*

<https://www.databricks.com/resources/ebook/mit-cio-generative-ai-report>

55. *PEFT*

<https://huggingface.co/docs/peft/index>

56. *Low-Rank Adaptation of Large Language Models (LoRA)*

<https://huggingface.co/docs/diffusers/training/lora>

57. *CustomGPT*

<https://docs.customgpt.ai/docs/welcome>

58. *Sintaxe*

<https://pt.wikipedia.org/wiki/Sintaxe>

59. *Semântica*

<https://pt.wikipedia.org/wiki/Sem%C3%A2ntica>

60. *Philosophical Investigations. Ludwig Wittgenstein, published posthumously in 1953.*

[https://en.wikipedia.org/wiki/Philosophical\\_Investigations](https://en.wikipedia.org/wiki/Philosophical_Investigations)

61. *Distributional semantics*

[https://en.m.wikipedia.org/wiki/Distributional\\_semantics](https://en.m.wikipedia.org/wiki/Distributional_semantics)

62. *n-gram*

<https://en.m.wikipedia.org/wiki/N-gram>

63. *Word n-gram language model*

[https://en.m.wikipedia.org/wiki/Word\\_n-gram\\_language\\_model](https://en.m.wikipedia.org/wiki/Word_n-gram_language_model)

64. *Recurrent Neural Network (RNN)*

[https://en.m.wikipedia.org/wiki/Recurrent\\_neural\\_network](https://en.m.wikipedia.org/wiki/Recurrent_neural_network)

65. *A Few Useful Things to Know About Machine Learning*. Pedro Domingos.

<https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf>

66. *Byte pair encoding*

[https://en.m.wikipedia.org/wiki/Byte\\_pair\\_encoding](https://en.m.wikipedia.org/wiki/Byte_pair_encoding)

67. *Tokenizer*

<https://platform.openai.com/tokenizer>

68. *Word embedding*

[https://en.m.wikipedia.org/wiki/Word\\_embedding](https://en.m.wikipedia.org/wiki/Word_embedding)

69. *Similaridade por cosseno*

[https://pt.wikipedia.org/wiki/Similaridade\\_por\\_cosseno](https://pt.wikipedia.org/wiki/Similaridade_por_cosseno)

70. *Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings*

<https://arxiv.org/abs/1607.06520>

71. *Word2vec*

<https://en.m.wikipedia.org/wiki/Word2vec>

72. *Cross-entropy*

<https://en.wikipedia.org/wiki/Cross-entropy>

73. *Softmax function*

[https://en.wikipedia.org/wiki/Softmax\\_function](https://en.wikipedia.org/wiki/Softmax_function)

74. *Softmax Activation Function — How It Actually Works*

<https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78>

75. *Temperature — LLMs*

[https://medium.com/@amansinghalm\\_33304/temperature-llms-b41d75870510](https://medium.com/@amansinghalm_33304/temperature-llms-b41d75870510)

76. *Turing test*

[https://en.wikipedia.org/wiki/Turing\\_test](https://en.wikipedia.org/wiki/Turing_test)

77. *ChatGPT broke the Turing test — the race is on for new ways to assess AI*

<https://www.nature.com/articles/d41586-023-02361-7>

78. *Cheat Sheet: Mastering Temperature and Top\_p in ChatGPT API*

<https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api-a-few-tips-and-tricks-on-controlling-the-creativity-deterministic-output-of-prompt-responses/172683>

79. *Preços do Serviço OpenAI Azure*

<https://azure.microsoft.com/pt-br/pricing/details/cognitive-services/openai-service/>

80. *Attention Is All You Need. Ashish Vaswani et al.*

<https://arxiv.org/abs/1706.03762>

81. *The NYTimes reporter @aatishb trained an AI to learn to write in the style of William Shakespeare*

<https://youtu.be/-dsZYeOuK9I?feature=shared>

82. *Llama 2: Open Foundation and Fine-Tuned Chat Models*

<https://arxiv.org/pdf/2307.09288.pdf>

83. *Training language models to follow instructions with human feedback*

<https://arxiv.org/pdf/2203.02155.pdf>

84. *InstructGPT*

<https://openai.com/research/instruction-following>

85. *Chatbot Arena Leaderboard Updates (Week 2) by: LMSYS Org, May 10, 2023*

<https://lmsys.org/blog/2023-05-10-leaderboard/>

86. *Prompt engineering*

[https://en.wikipedia.org/wiki/Prompt\\_engineering](https://en.wikipedia.org/wiki/Prompt_engineering)

87. *O que é engenharia por prompt?*

<https://aws.amazon.com/pt/what-is/prompt-engineering/>

88. *Best practices for prompt engineering with OpenAI API*

<https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api>

89. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Jason Wei et al.*

<https://arxiv.org/abs/2201.11903>

90. *Arithmetic Reasoning on GSM8K*

<https://paperswithcode.com/sota/arithmetic-reasoning-on-gsm8k>

91. *ChatGPT plugins*

<https://openai.com/blog/chatgpt-plugins>

92. *Neural Networks and Deep Learning: A Textbook 2nd ed. 2023 Edition. Charu C. Aggarwal*

<https://a.co/d/0K4Tmmw>

93. *Deep Learning - Ian Goodfellow*

<https://a.co/d/fnbh98d>

94. *Método do gradiente*

[https://pt.wikipedia.org/wiki/M%C3%A9todo\\_do\\_gradiente](https://pt.wikipedia.org/wiki/M%C3%A9todo_do_gradiente)

95. *Dive into Deep Learning - Interactive deep learning book with code, math, and discussions. Aston Zhang et al.*

<https://d2l.ai/index.html>

96. *Encoder-Decoder Models for Natural Language Processing. Daniel Ibanez*

<https://www.baeldung.com/cs/nlp-encoder-decoder-models>

97. *Gated recurrent unit (GRU)*

<https://en.m.wikipedia.org/>

98. *Long short-term memory*

[https://en.m.wikipedia.org/wiki/Long\\_short-term\\_memory](https://en.m.wikipedia.org/wiki/Long_short-term_memory)

99. *RNN vs GRU vs LSTM. Hemanth Pedamallu*

<https://medium.com/analytics-vidhya/rnn-vs-gru-vs-lstm-863b0b7b1573>

100. *Convolutional neural network*

[https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network](https://en.wikipedia.org/wiki/Convolutional_neural_network)

101. *Rectifier (neural networks)*

[https://en.wikipedia.org/wiki/Rectifier\\_\(neural\\_networks\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))

102. *Neural machine translation by jointly learning to align and translate.* Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio

<https://arxiv.org/abs/1409.0473>

103. *Attention is all you need; Attentional Neural Network Models | Łukasz Kaiser | Masterclass*

<https://youtu.be/rBCqOTefxvg?feature=shared>

104. *How GPT3 Works - Visualizations and Animations.* Jay Alammar

<https://jalammar.github.io/how-gpt3-works-visualizations-animations/>

105. *The Illustrated Transformer.* Jay Alammar

<https://jalammar.github.io/illustrated-transformer/>

106. *Transformer (machine learning model)*

[https://en.m.wikipedia.org/wiki/Transformer\\_\(machine\\_learning\\_model\)](https://en.m.wikipedia.org/wiki/Transformer_(machine_learning_model))

107. *Attention (machine learning)*

[https://en.m.wikipedia.org/wiki/Attention\\_\(machine\\_learning\)](https://en.m.wikipedia.org/wiki/Attention_(machine_learning))

108. *Transformer Architecture - Lei Mao*

<https://leimao.github.io/blog/Transformer-Explained/>

109. *Produto escalar*

[https://pt.wikipedia.org/wiki/Produto\\_escalar](https://pt.wikipedia.org/wiki/Produto_escalar)

110. *Transformers: The best idea in AI | Andrej Karpathy and Lex Fridman*

<https://youtu.be/9uw3F6rndnA?feature=shared>

111. *Understanding the Adam optimizer for gradient descent*

<https://www.linkedin.com/pulse/understanding-adam-optimizer-gradient-descent-evan-dunbar/>

112. *Dropout in Neural Networks*

<https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9>

113. *What is Label Smoothing?*

<https://towardsdatascience.com/what-is-label-smoothing-108debd7ef06>

114. *The Unpredictable Abilities Emerging From Large AI Models. Stephen Ornes. Quanta Magazine.*

<https://www.quantamagazine.org/the-unpredictable-abilities-emerging-from-large-ai-models-20230316/>



115. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.* Alexey Dosovitskiy et al.  
<https://arxiv.org/abs/2010.11929>

116. *Will Transformers Take Over Artificial Intelligence?* Stephen Ornes, *Quanta Magazine*.  
<https://www.quantamagazine.org/will-transformers-take-over-artificial-intelligence-20220310/>

117. *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)*  
<https://www.image-net.org/challenges/LSVRC/>

118. *Predictability and Surprise in Large Generative Models.* Deep Ganguli et al. Anthropic, USA  
<https://dl.acm.org/doi/abs/10.1145/3531146.3533229>

119. *The Capacity for Moral Self-Correction in Large Language Models*  
<https://www.anthropic.com/index/the-capacity-for-moral-self-correction-in-large-language-models>

120. *Ética na Inteligência Artificial - Regulamentações*  
<https://www.etica-ia.com/leis>