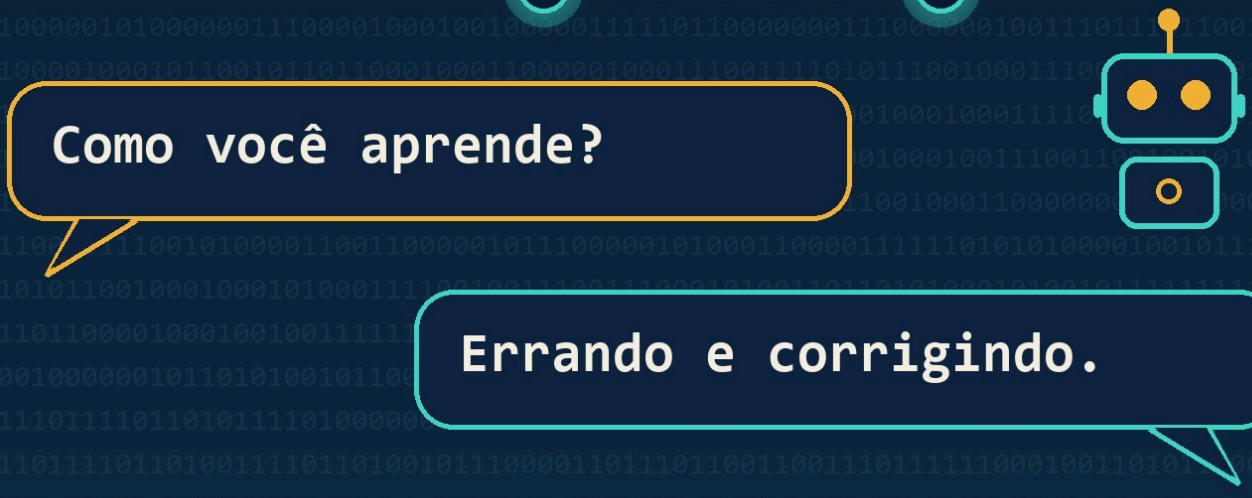
IA

A MÁQUINA QUE APRENDE

Da primeira conta ao chat:
inteligência artificial e Python
para quem nunca programou





BELISÁRIO RETTO DE ABREU



IA — A Máquina que Aprende

BELISÁRIO RETTO DE ABREU

Novembro de 2026

Texto escrito com a ajuda da inteligência artificial Claude. Ilustrações vetoriais.
Também publicado em multiversodigital.com.br/Biografia

Sumário

Apresentação
Capítulo 1: Regra ou aprendizado?
Capítulo 2: Python em 15 minutos
Capítulo 3: Tudo vira número
Capítulo 4: Aprender é errar e corrigir
Capítulo 5: O neurônio
Capítulo 6: A rede
Capítulo 7: Ensinar a máquina a ver
Capítulo 8: Palavras viram números
Capítulo 9: Prever a próxima palavra
Capítulo 10: A atenção
Capítulo 11: Do modelo ao assistente
Capítulo 12: Um chat de verdade
Glossário
Como instalar e rodar tudo

APRESENTAÇÃO



Este livro é para quem nunca programou e quer entender, sem mistério, como uma máquina aprende e como um assistente de inteligência artificial consegue conversar. Você não precisa saber matemática além da tabuada, nem ter um computador potente. Precisa de curiosidade e de um computador comum, com internet para instalar o Python.

O caminho é um só, do começo ao fim: do primeiro programa de três linhas até um chat que guarda a conversa e responde. Em cada capítulo você vai ler uma ideia, ver um desenho dela e rodar um exemplo em Python. Os exemplos são curtos, de 10 a 30 linhas, e todos foram executados antes de entrar aqui: os resultados que aparecem no livro são os que o computador realmente imprimiu.



Quem é o Bit

Quem vai com você é o Bit, um robozinho curioso. Ele erra, corrige, fica surpreso e, às vezes, inventa coisa com muita confiança, como os assistentes de verdade. Esse é o jeito mais honesto que encontrei de mostrar o que uma máquina que aprende faz bem e o que ela faz mal.



Como o livro está organizado

São doze capítulos. Os três primeiros preparam o terreno: a diferença entre dar regras e dar exemplos, o Python em quinze minutos e a ideia de que tudo vira número. Do quarto ao sexto, a máquina aprende de verdade: erra, corrige, ganha um neurônio e depois uma rede. No sétimo ela aprende a ver. Do oitavo ao décimo primeiro entramos na linguagem: palavras viram números, a máquina prevê a próxima palavra, aprende a prestar atenção e se torna um assistente. O último capítulo monta o chat e fala dos cuidados que um bom usuário precisa ter.

No fim há um glossário e um apêndice que ensina a instalar e rodar tudo. Se algo emperrar, comece por ele.

E há um atalho: todos os exemplos rodam também no site, no laboratório de Python (multiversodigital.com.br/Biografia/python-lab/), sem instalar nada. Você escolhe o exemplo, aperta “Rodar”, muda um número e roda de novo.



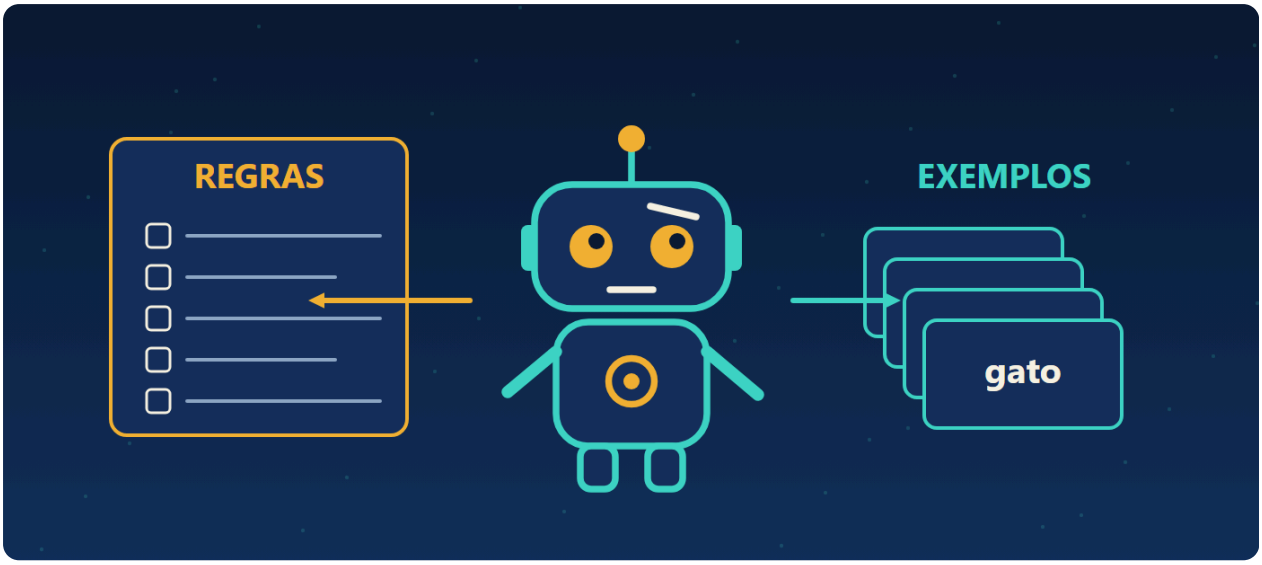
Uma palavra sobre honestidade

Este livro foi escrito com o auxílio de inteligência artificial, como todo o trabalho do Multiverso Digital: eu digo isso abertamente, sem esconder. As ideias, as escolhas e a responsabilidade pelo que está escrito são do autor; os exemplos de código foram testados na máquina, e os fatos históricos citados são poucos e foram conferidos. Onde um exemplo é só uma ilustração, o livro avisa.

Ler sobre IA e usar IA para escrever sobre IA tem a sua ironia, e eu aproveito. Boa viagem. O Bit já está esperando.



CAPÍTULO 1: REGRA OU APRENDIZADO?



O Bit chegou à sala de um jeito desajeitado e perguntou:

— Como é que se ensina uma máquina?

A resposta curta é que existem duas maneiras, e a diferença entre elas é a chave de todo este livro.

A maneira antiga: escrever as regras

Durante décadas, programar foi isto: uma pessoa pensa em tudo o que a máquina deve fazer e escreve, passo a passo, as regras. Se acontecer isso, faça aquilo. Funciona muito bem quando o problema é claro. Calcular o troco, somar uma fatura e ordenar uma lista de nomes são tarefas em que quem escreve as regras sabe exatamente o que quer.

Mas imagine que você precisa ensinar a máquina a separar as mensagens que você quer ler das mensagens de golpe, o spam. Como se escreve a regra? Você pensa um pouco e chega a algo assim: se a mensagem tem as palavras “grátis”, “ganhe” ou “clique aqui”, é spam.



Figura 1.1 · Duas maneiras de fazer a máquina responder.

Rode você mesmo: o filtro de regras

Este é o primeiro exemplo do livro. Não se assuste com o código: no próximo capítulo você aprende o Python com calma. Por ora, leia como se fosse uma receita. Há uma lista de palavras suspeitas e uma função que olha a mensagem e diz verdadeiro (True) se achar alguma delas.



c01_spam_regras.py



```
# Um filtro de spam feito de regras escritas à mão.
PALAVRAS_SUSPEITAS = ["grátis", "ganhe", "promoção", "clique aqui", "urgente"]

def parece_spam(mensagem):
    texto = mensagem.lower()
    for palavra in PALAVRAS_SUSPEITAS:
        if palavra in texto:
            return True
    return False

mensagens = [
    "Ganhe um celular grátis, clique aqui!",
    "Vovó, o almoço de domingo é às 12h.",
    "Sua conta foi bloqueada. Confirme seus dados agora.",
]
for m in mensagens:
    print(parece_spam(m), "->", m)
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c01_spam_regras

Quando o computador roda esse programa, ele imprime isto:

o computador imprime

```
True -> Ganhe um celular grátis, clique aqui!
False -> Vovó, o almoço de domingo é às 12h.
False -> Sua conta foi bloqueada. Confirme seus dados agora.
```

A primeira mensagem foi pega, e acertamos. A segunda, da vovó, passou, e também acertamos. Mas olhe a terceira: “Sua conta foi bloqueada. Confirme seus dados agora.” É um golpe clássico, e o filtro deixou passar, porque nenhuma das nossas palavras aparece ali.

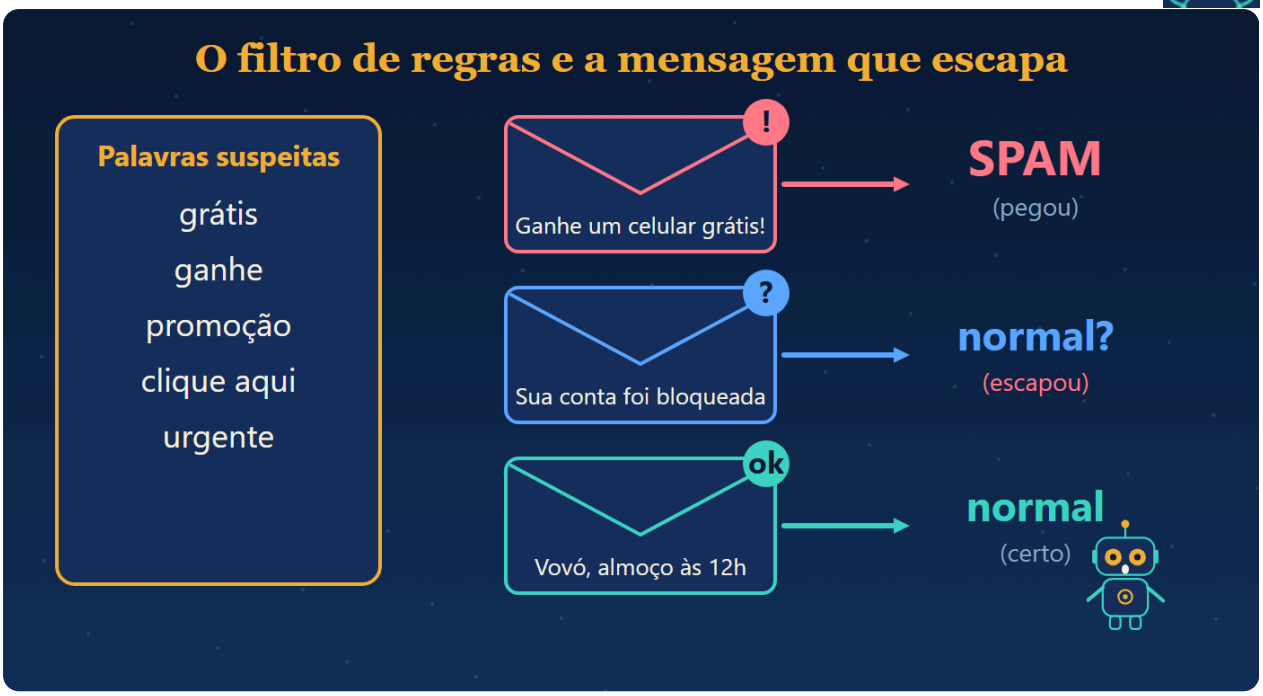


Figura 1.2 · O filtro de regras pega o que a lista previu e perde o que ninguém previu.



O problema não é a pressa, é o infinito

Dá para consertar? Dá. Basta acrescentar “bloqueada” e “confirme seus dados” à lista. Na semana seguinte, os golpistas mudam o texto. Você acrescenta mais palavras. Elas mudam de novo. Cada regra nova pode atrapalhar mensagens boas: uma mensagem do seu banco, de verdade, também pode dizer “confirme seus dados”. A lista cresce, vira uma maçaroca e ninguém mais sabe por que o filtro decidiu o que decidiu.

Esse é o limite das regras escritas à mão: o mundo é mais variado do que a imaginação de quem escreve.



A maneira nova: mostrar exemplos

E se, em vez de escrever as regras, você mostrasse à máquina mil mensagens já separadas por uma pessoa, “isto é spam, isto não é”, e deixasse que ela descobrisse sozinha o que costuma aparecer em cada grupo? Foi essa a virada. No lugar de entregar dados mais regras e receber a resposta, entregamos os dados mais as respostas certas e recebemos as regras.



Chamamos isso de aprendizado de máquina. Parece truque, mas é só uma troca de papéis: a pessoa deixa de ser a autora das regras e passa a ser a professora que corrige. As regras que a máquina descobre ficam guardadas em números, que veremos nos próximos capítulos.

Um pouco de história, sem decorar nada

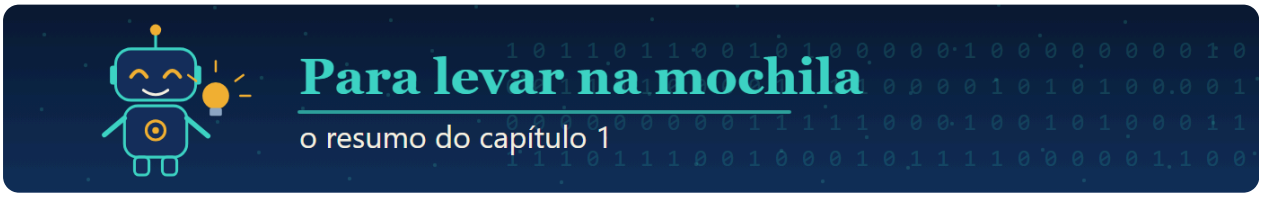
A expressão “inteligência artificial” foi proposta por John McCarthy e colegas para um encontro de verão em Dartmouth, nos Estados Unidos, em 1956. Pouco depois, em 1959, Arthur Samuel descreveu um programa que aprendia a jogar damas jogando contra si mesmo, e usou a expressão “aprendizado de máquina”. Em 1966, Joseph Weizenbaum criou a ELIZA, que conversava imitando uma psicoterapeuta, mas por regras de substituição de frases: quem escrevia “estou triste” recebia de volta algo como “por que você está triste?”. Muita gente se abriu com ela, apesar de a ELIZA não entender nada.



Figura 1.3 · Seis marcos que vamos reencontrar nos próximos capítulos.

Os outros marcos da linha do tempo, a retropropagação de 1986, o Transformer de 2017 e o ChatGPT de 2022, aparecem mais adiante, cada um no seu capítulo.

O que o Bit aprendeu

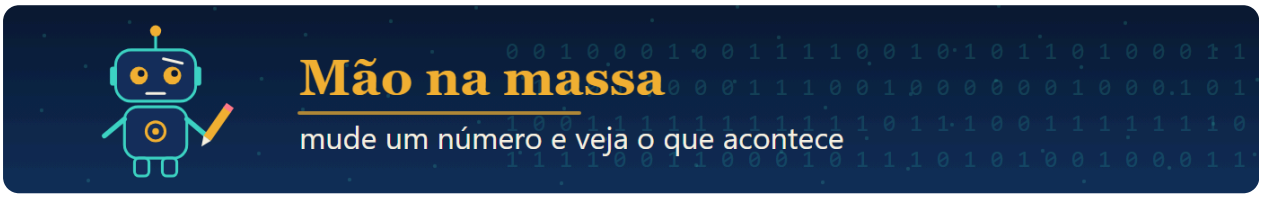




O Bit anotou três coisas no caderno dele:

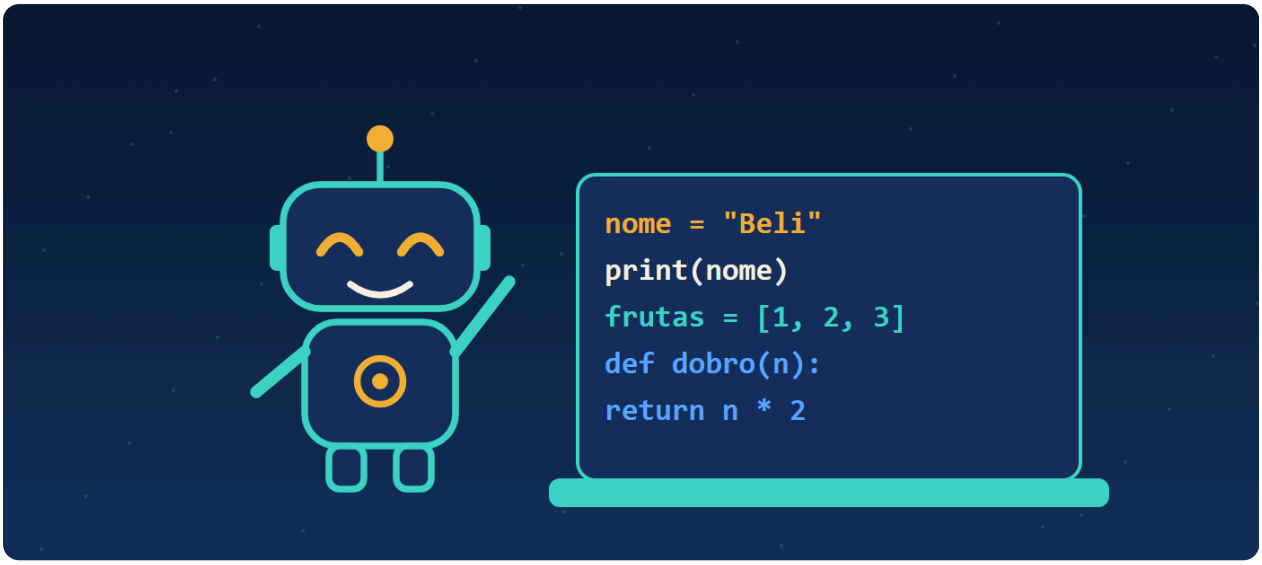
- Regras escritas à mão funcionam quando o problema é claro, e falham quando o mundo é variado.
- Aprender é descobrir as regras a partir de exemplos com a resposta certa.
- Quem ensina, ensina por exemplos. Por isso a qualidade dos exemplos importa tanto, tema que voltará várias vezes.

Experimente



Pegue o filtro de regras e acrescente “bloqueada” à lista de palavras suspeitas. Rode de novo. A terceira mensagem agora é pega? E a mensagem “Sua escola foi bloqueada pela chuva, volte amanhã” também? Pense: o que você teria de fazer para o filtro distinguir as duas?

CAPÍTULO 2: PYTHON EM 15 MINUTOS



Python é uma linguagem para dar ordens ao computador. Foi criada por Guido van Rossum e lançada em 1991, e ficou famosa por um motivo simples: lê-se quase como português mal escrito. É por isso que quase toda a inteligência artificial de hoje é feita com ela. Você vai aprender aqui só o que precisa para o resto do livro: variáveis, listas, repetição e funções.



Antes de começar: instalar e rodar

Se preferir não instalar nada por enquanto, use o laboratório de Python do site (multiversodigital.com.br/Biografia/python-lab/): ele roda os exemplos do livro no navegador, e dá para mudar o código e rodar de novo. Na versão do livro na tela, cada exemplo traz um quadro para rodar ali mesmo.

O Python é gratuito. No Windows, baixe o instalador em python.org e, na primeira tela, marque a caixa “Add python.exe to PATH” antes de instalar. Depois abra o “Prompt de Comando” (ou o Terminal) e digite `python --version`. Se aparecer um número, como 3.13, está pronto. O apêndice no fim do livro detalha cada passo, inclusive para Mac e Linux.

Há duas maneiras de rodar um programa. Você pode digitar `python` e escrever linha por linha, ou guardar as linhas num arquivo, por exemplo `ola.py` escrito no Bloco de Notas, e rodar `python ola.py`. Neste livro, sempre o arquivo: assim você guarda o que fez. Uma dica para o Windows: se os acentos saírem estranhos na tela, rode `python -X utf8 ola.py`.

Variável: uma caixa com etiqueta

Uma variável guarda um valor com um nome. O sinal de igual não significa “é igual a”, como na matemática: significa “guarde isto na caixa com este nome”.

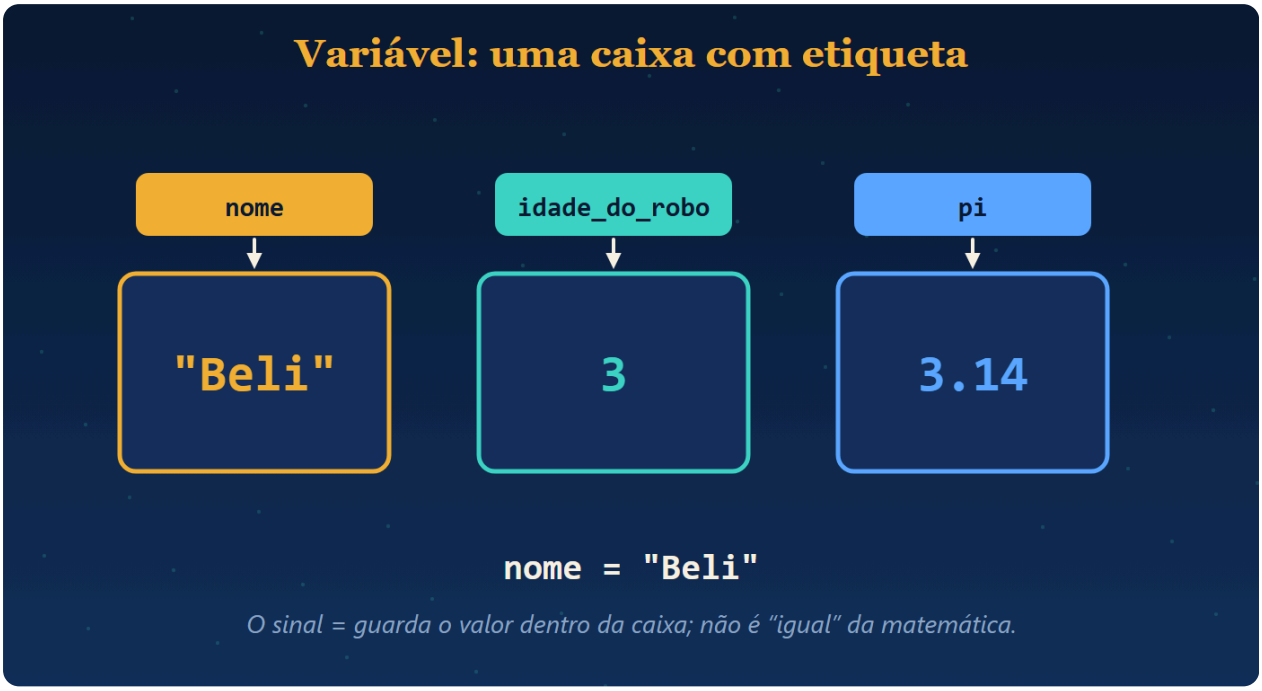


Figura 2.1 · Variáveis são caixas com etiqueta.

Lista: gavetas numeradas

Uma lista guarda vários valores, em ordem. Cada gaveta tem um número, e a contagem começa no zero. Esse é o tropeço mais comum de quem começa: a primeira fruta é a de número 0, não a de número 1.

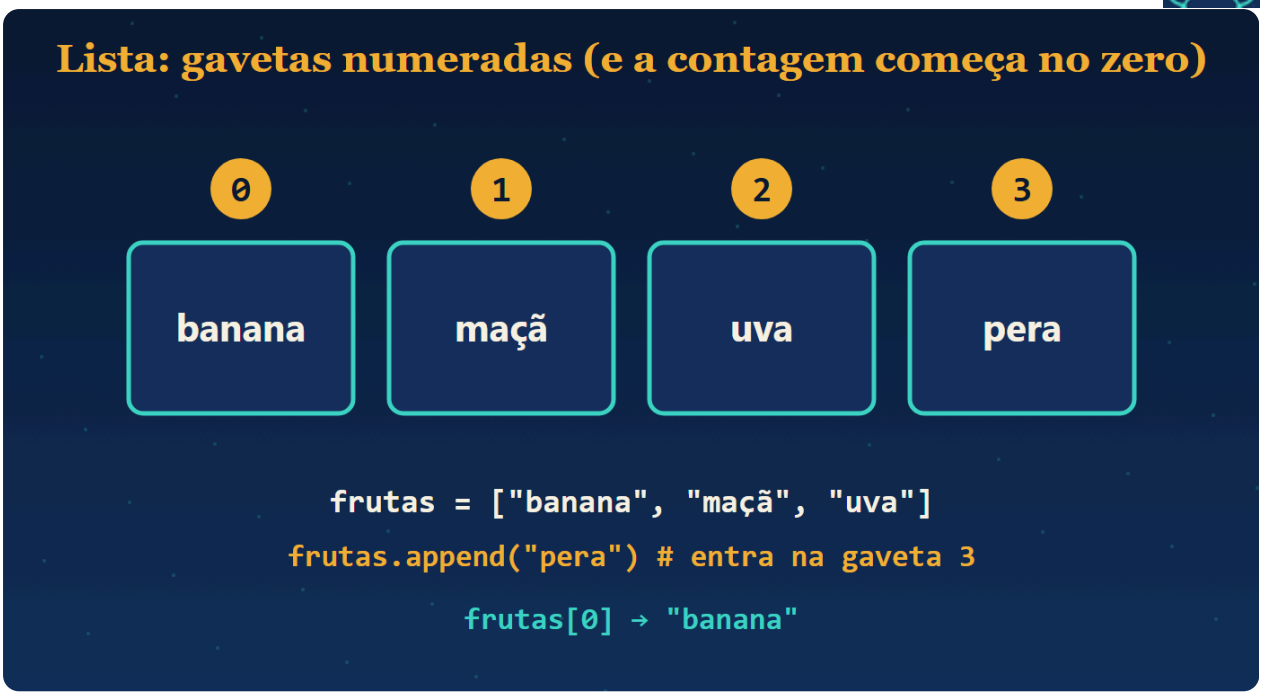


Figura 2.2 · A lista guarda os valores em ordem, a partir do zero.

Repetição: a esteira

O computador não cansa. Por isso a ideia mais útil é mandar repetir. O comando `for` pega cada item da lista, um de cada vez, e executa o que estiver recuado logo abaixo. O recuo, quatro espaços, faz parte da linguagem: é assim que o Python sabe o que pertence à repetição.



Figura 2.3 · O for pega um item por vez.

Função: uma máquina com nome

Uma função é um pedaço de programa com nome. Você escreve uma vez e usa quantas quiser. Ela recebe um valor, faz uma conta e devolve o resultado com a palavra `return`.

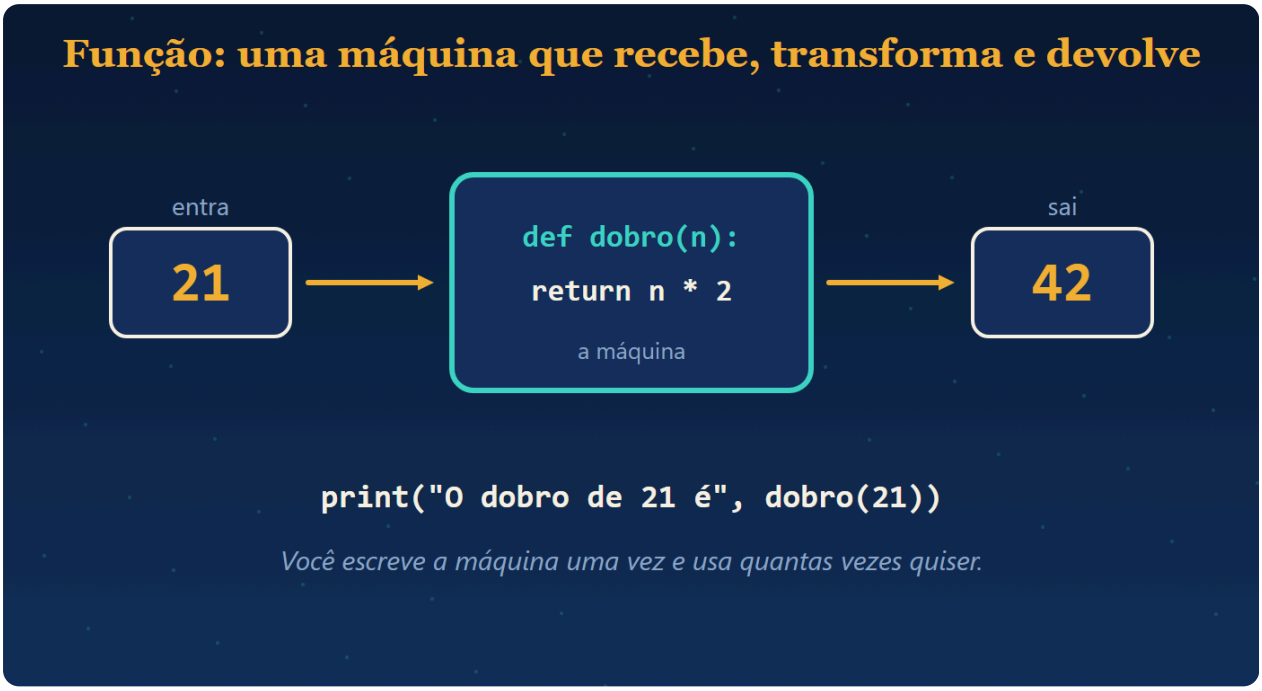


Figura 2.4 · Uma função recebe, transforma e devolve.

Rode você mesmo

Junte tudo num arquivo chamado `c02_python.py`. A primeira linha começa com o sinal # (a cerquilha, ou "jogo da velha"): é um comentário, que o Python ignora. Serve para você, não para a máquina.



c02_python.py



```
# Python em poucas linhas: variável, lista, repetição, função.
nome = "Beli"
idade_do_robo = 3
print("Olá,", nome)

frutas = ["banana", "maçã", "uva"]
frutas.append("pera")
for fruta in frutas:
    print("Eu gosto de", fruta)

def dobro(numero):
    return numero * 2

print("O dobro de 21 é", dobro(21))
print("Tenho", len(frutas), "frutas")
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c02_python

o computador imprime

```
Olá, Beli
Eu gosto de banana
Eu gosto de maçã
Eu gosto de uva
Eu gosto de pera
O dobro de 21 é 42
Tenho 4 frutas
```

Duas coisas para notar. A função `len` (de “length”, comprimento) conta quantos itens a lista tem. E o `append` acrescentou a pera no fim, razão pela qual são quatro frutas e não três. Se você chegou até aqui, já sabe tudo o que o resto do livro exige de linguagem. O que vem depois é só combinar essas quatro peças de jeitos novos.



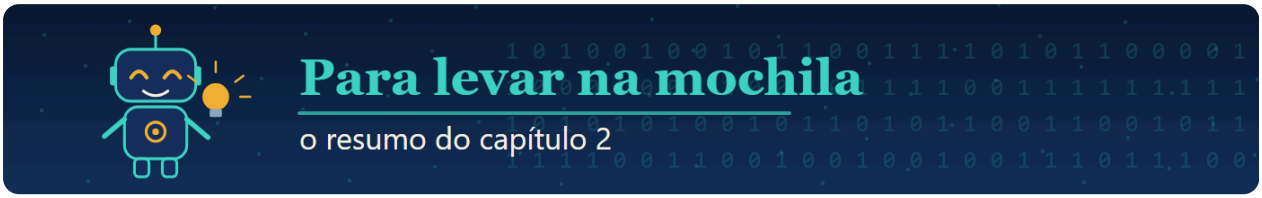
Quando der erro (e vai dar)

Todo mundo erra ao programar, e o Python avisa em vermelho o que não entendeu. Os erros mais comuns do começo são três: esquecer os dois-pontos depois do `for` ou do `def`; misturar o recuo (quatro espaços numa linha, três em outra); e escrever o nome da variável de um jeito num lugar e de outro jeito em outro. A mensagem de erro costuma dizer o número da linha. Leia a última linha dela primeiro: é a que diz o que houve.



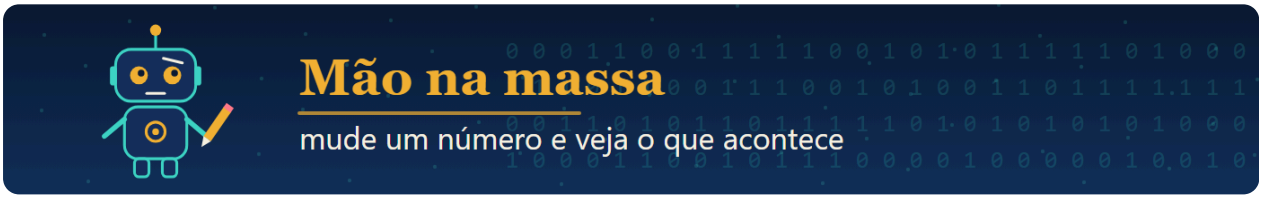
E um conselho do Bit, que erra para viver: erro não é fracasso, é informação. É exatamente assim que a máquina vai aprender nos próximos capítulos.

O que o Bit aprendeu



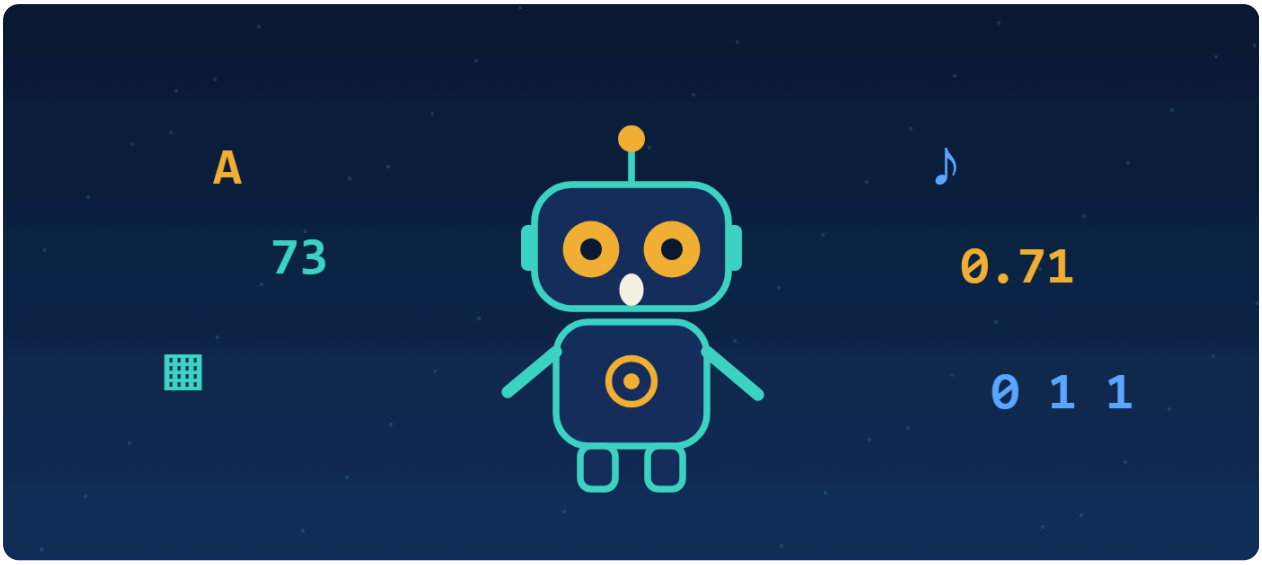
- Variável guarda um valor com um nome.
- Lista guarda vários valores em ordem, contando a partir do zero.
- O `for` repete uma ação para cada item.
- Função é uma máquina com nome: recebe, transforma e devolve.

Experimente



Troque “Beli” pelo seu nome. Acrescente duas frutas à lista. Crie uma função chamada `triplo` e chame-a com o número que você quiser. Depois, de propósito, apague os dois-pontos do `for` e leia a mensagem de erro: ela está dizendo exatamente onde você escorregou.

CAPÍTULO 3: TUDO VIRA NÚMERO



Uma máquina de calcular só entende de números. Então como é que um computador mostra uma foto, toca uma música ou lê o seu texto? A resposta é a mais importante deste livro até agora: ele não faz nada disso. O que ele faz é transformar tudo em número e trabalhar com os números.

Texto: cada letra tem um número

Quando você aperta a tecla “I”, o computador guarda o número 73. Para o “A”, o 65. Essa tabela de combinações, em que cada letra tem o seu número, foi combinada há décadas entre os fabricantes, para que os computadores se entendessem. A mesma ideia, ampliada para os acentos, o “ç” e até emojis, é a que está no seu celular.

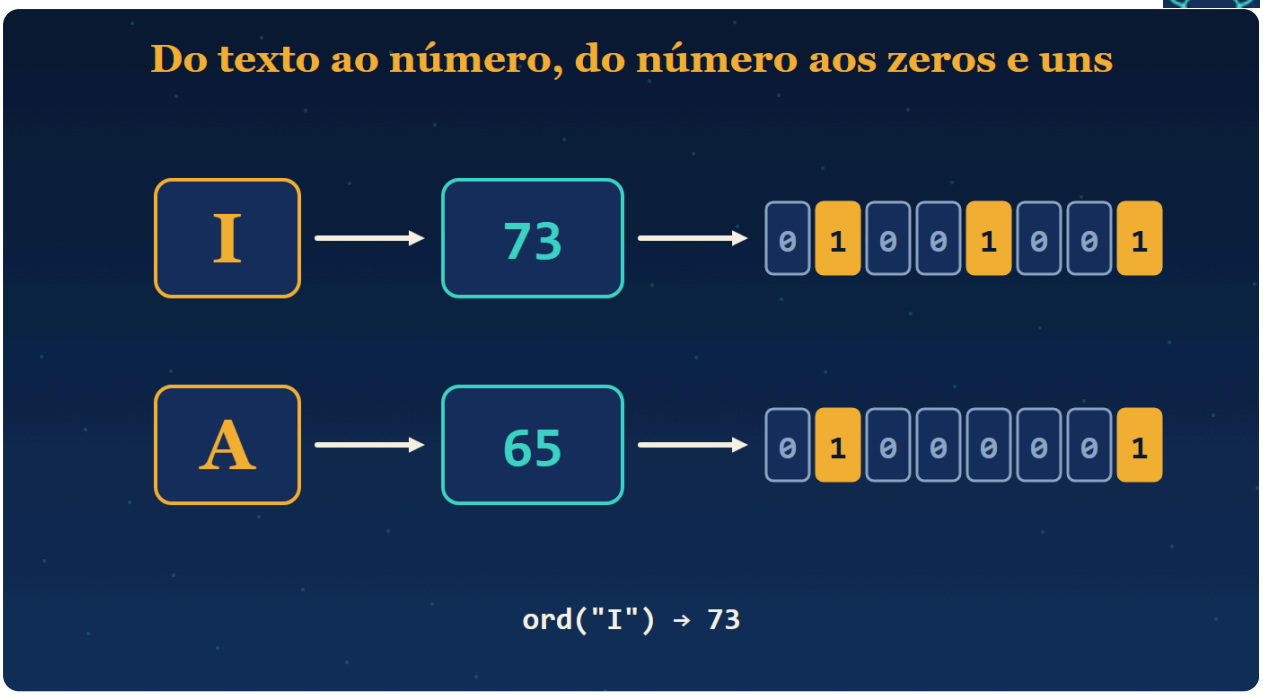


Figura 3.1 · Da letra ao número, do número aos zeros e uns.

E o número, por dentro, é uma sequência de zeros e uns, os bits. O 73 vira 01001001. Cada bit é um interruptor ligado ou desligado. Do interruptor ao texto inteiro, só existe empilhamento.

Imagem: uma grade de números

Uma foto é uma grade de pontinhos, os pixels. Em cada pixel, um número diz quão claro ele é. Uma imagem em preto e branco muito simples de 5 por 5 pontos, com 1 para aceso e 0 para apagado, já pode desenhar um número.



Figura 3.2 · A imagem que você vê e os números que a máquina recebe.

Foto colorida? Mesma ideia, com três números por ponto: quanto tem de vermelho, de verde e de azul, de 0 a 255 cada um. O dourado do robozinho Bit é (244, 178, 52): muito vermelho, bastante verde, pouco azul.



Figura 3.3 · Cor são três números.

Som: a altura da onda

O som é uma vibração do ar. Um microfone mede, milhares de vezes por segundo, o quanto o ar está empurrado ou puxado naquele instante, e cada medida é um número. A música no seu celular é uma lista imensa de números que sobem e descem.

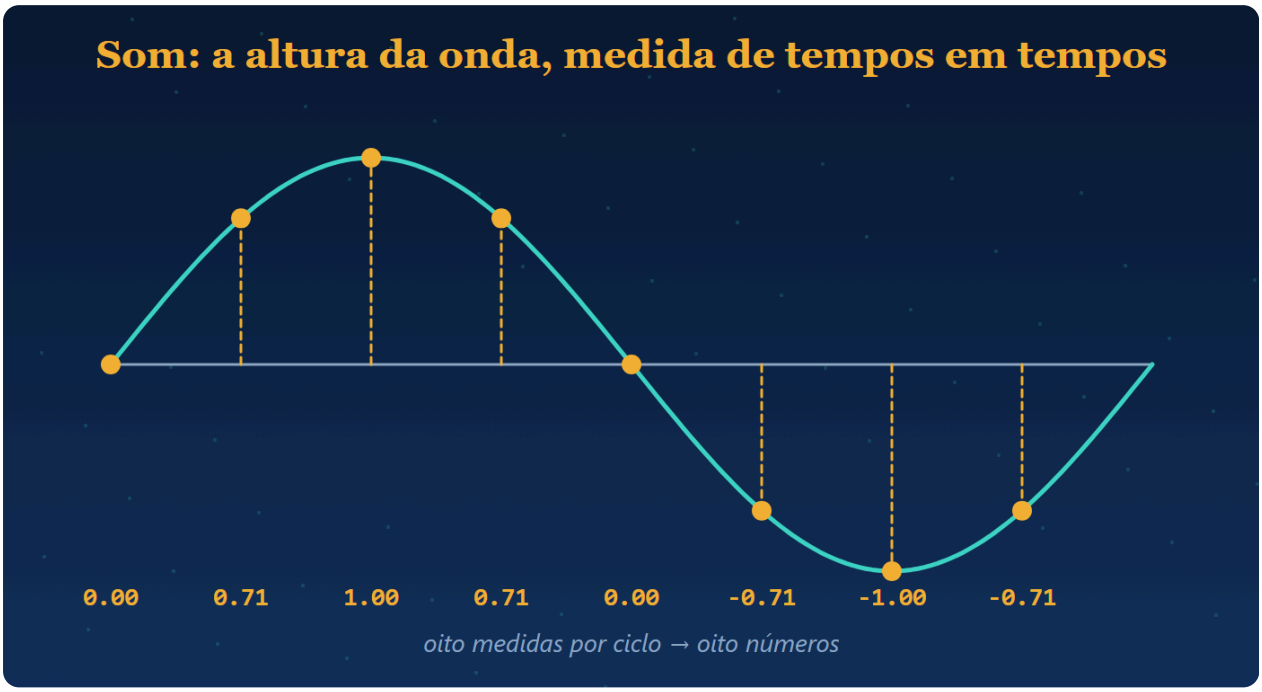


Figura 3.4 · Oito medidas de uma onda: o som vira uma lista de números.

Rode você mesmo

O programa abaixo faz as três coisas. Ele usa um pedaço novo do Python: a função `ord`, que dá o número de uma letra, e o módulo `math`, que traz o seno, a matemática da onda. Não se preocupe em entender cada detalhe da conta; o que importa é ver o resultado.



c03_numeros.py



```
# Tudo vira número: texto, imagem e som.
import math

# 1) Texto: cada letra tem um número.
for letra in "IA":
    print(letra, "->", ord(letra))
print("oi em bytes:", list("oi".encode("utf-8")))

# 2) Imagem: uma grade de 5x5, 0 = apagado, 1 = aceso.
imagem = [
    [0, 0, 1, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 1, 1, 1, 0],
]
for linha in imagem:
    print("".join("#" if p else "." for p in linha), linha)

# 3) Som: uma onda medida 8 vezes por ciclo.
onda = [round(math.sin(2 * math.pi * i / 8), 2) for i in range(8)]
print("som:", onda)
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c03_numeros

o computador imprime

```
I -> 73
A -> 65
oi em bytes: [111, 105]
..#.. [0, 0, 1, 0, 0]
.##.. [0, 1, 1, 0, 0]
..#.. [0, 0, 1, 0, 0]
..#.. [0, 0, 1, 0, 0]
.###. [0, 1, 1, 1, 0]
som: [0.0, 0.71, 1.0, 0.71, 0.0, -0.71, -1.0, -0.71]
```

A letra I é 73 e a A é 65, os mesmos números do desenho. A imagem do “1” apareceu duas vezes, uma desenhada com cerquilhas e pontos, outra como números. E o som são oito medidas, que sobem até 1, descem até −1 e voltam.

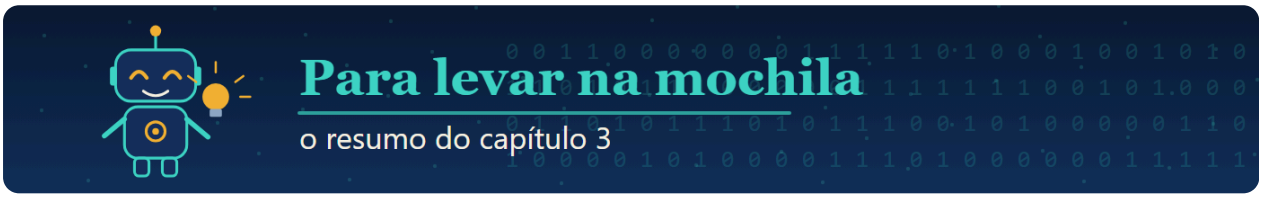


Por que isso importa para aprender

Porque agora tudo tem a mesma forma: listas de números. Uma foto de um gato, uma frase, um áudio, a idade de uma pessoa, tudo cabe numa lista. E contas com números a máquina faz muito bem, e muito depressa. A inteligência artificial é, no fundo, uma enorme quantidade de contas com números que aprenderam a ter sentido.

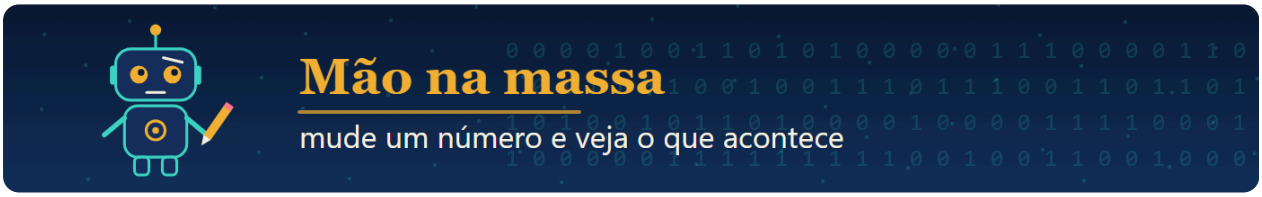
Há um preço. A máquina só enxerga o que foi traduzido para número. O que a tradução perdeu, ela nunca soube. Uma frase de ironia, um olhar, o cheiro de café: se não estiver nos números, não existe para ela. Guarde isso, que voltará no último capítulo.

O que o Bit aprendeu



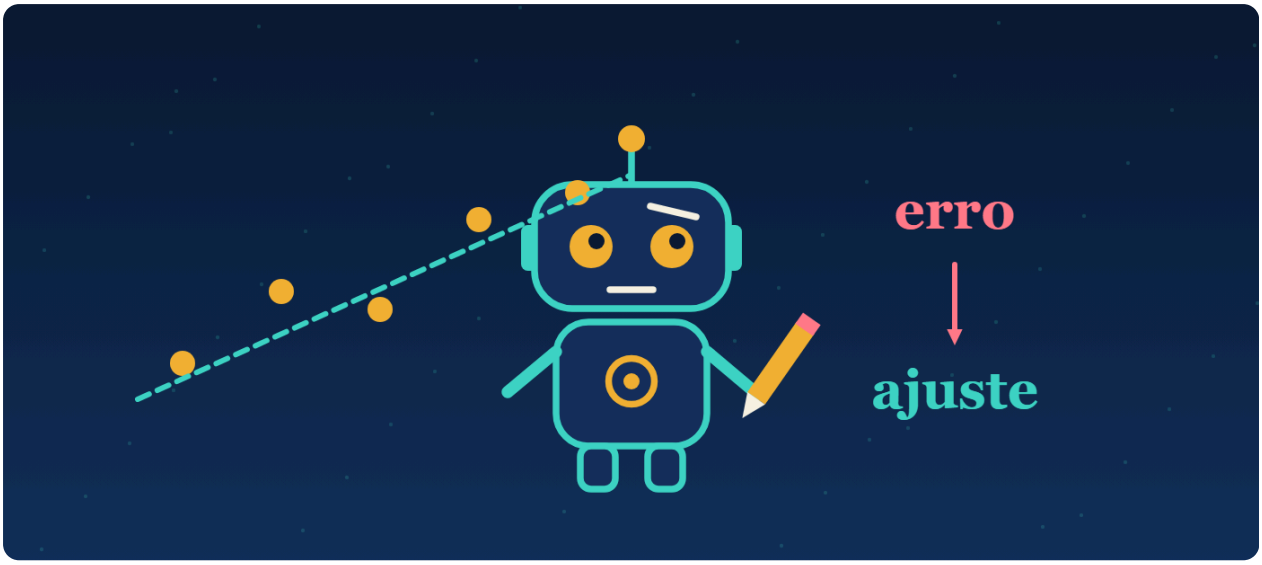
- Letras, imagens, cores e sons viram números.
- Por dentro, todo número é uma fila de zeros e uns.
- Se algo não foi traduzido em número, a máquina não enxerga.

Experimente



Troque as letras do programa por “ZAP” e veja os números. Desenhe um 7 numa grade de 5 por 5, trocando os zeros e uns da lista `imagem`. Mude `range(8)` para `range(16)` e `/ 8` para `/ 16` para medir a onda duas vezes mais vezes por ciclo.

CAPÍTULO 4: APRENDER É ERRAR E CORRIGIR



Chegamos ao coração do livro. Todo aprendizado de máquina, do filtro de spam ao assistente que conversa, usa a mesma ideia em tamanhos diferentes: faz um palpite, mede o erro, corrige um pouquinho e repete. Neste capítulo vamos ver isso acontecer num problema tão pequeno que cabe numa mão.



O problema: quanta nota vale uma hora de estudo?

Cinco alunos anotaram quantas horas estudaram e que nota tiraram, de 0 a 10. Eles estudaram 1, 2, 3, 4 e 5 horas e tiraram 3, 5, 6, 8 e 9. Queremos que a máquina descubra a regra, para poder prever a nota de quem estudar 6 horas.

Escolhemos uma regra do tipo mais simples possível: a nota é a vezes as horas, mais b. Em desenho, é uma reta. O a diz o quanto a nota sobe por hora estudada. O b diz a nota de quem não estuda nada. A máquina não sabe quanto valem a e b. O trabalho dela é descobrir.

O palpite e o erro

A máquina começa chutando: $a = 0$ e $b = 0$. Com isso, ela prevê nota zero para todo mundo. Péssimo palpite, e dá para medir o quanto: para cada aluno, a diferença entre a previsão e a nota verdadeira é o erro. Elevamos cada erro ao quadrado (assim erro para cima e para baixo não se cancelam, e erro grande pesa mais que erro pequeno) e tiramos a média. No começo, essa média dá 43.

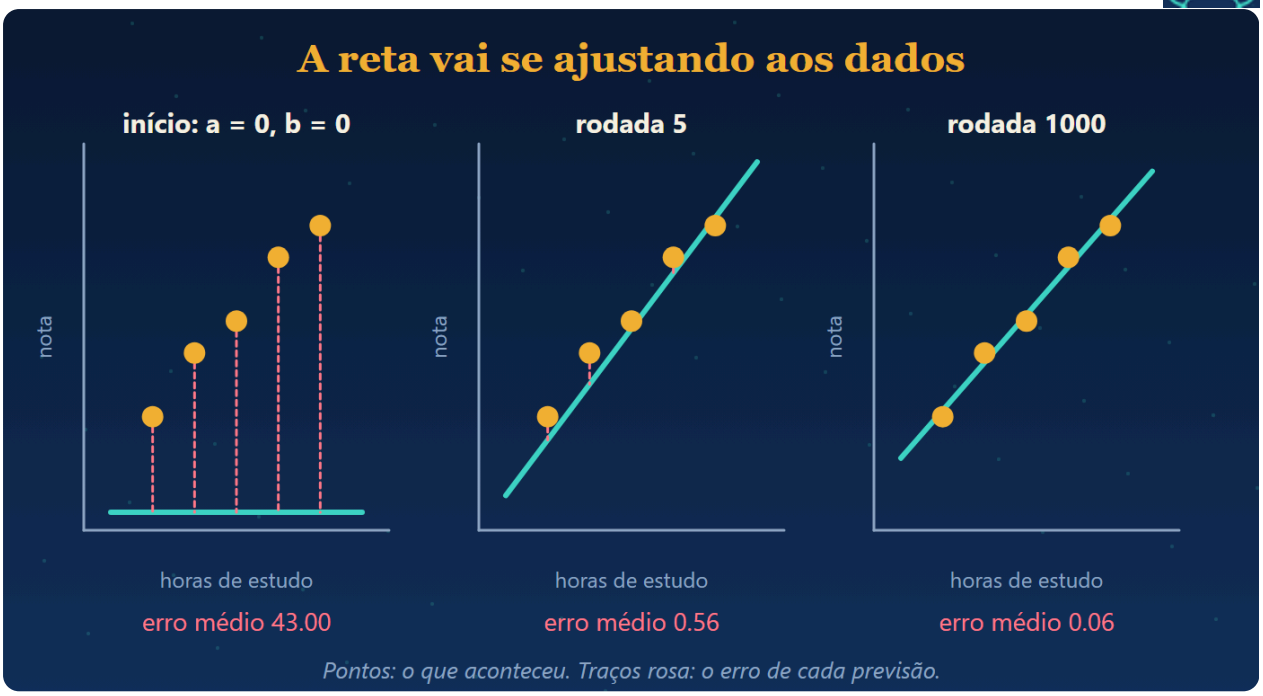


Figura 4.1 · A reta vai se ajustando aos dados. Os traços rosa são os erros.

Corrigir um pouquinho

Agora vem a parte esperta. Para cada aluno, perguntamos: se eu aumentasse um tiquinho o *a*, o erro aumentaria ou diminuiria? E o *b*? A resposta sai de uma conta de cálculo, a derivada, mas você não precisa dela para entender a ideia. Imagine que você está numa colina com neblina e quer descer até o vale. Você não vê o vale, mas sente, com o pé, para que lado o chão desce. Dá um passo nessa direção, sente de novo e dá outro passo.



Figura 4.2 · Descer o vale: a cada passo, o erro diminui.



Esse método tem nome, descida do gradiente, e é muito antigo: o matemático francês Augustin-Louis Cauchy já o descreveu em 1847. O tamanho do passo é o que chamamos de taxa de aprendizado. No nosso programa, é `passo = 0.02`.

Rode você mesmo



c04_reta.py

```
# A máquina aprende a nota a partir das horas de estudo, errando e corrigindo.
horas = [1, 2, 3, 4, 5]
notas = [3, 5, 6, 8, 9]          # o que aconteceu de verdade

a = 0.0      # quanto a nota sobe por hora (começa chutado)
b = 0.0      # nota de quem não estuda (começa chutado)
passo = 0.02 # o tamanho de cada correção

for rodada in range(1, 1001):
    erro_a = 0.0
    erro_b = 0.0
    erro_total = 0.0
    for h, n in zip(horas, notas):
        previsao = a * h + b
        erro = previsao - n
        erro_total += erro * erro
        erro_a += 2 * erro * h
        erro_b += 2 * erro
    a -= passo * erro_a / len(horas)
    b -= passo * erro_b / len(horas)
    if rodada in (1, 2, 5, 10, 50, 200, 1000):
        print(f"rodada {rodada:4d}: a={a:.2f}  b={b:.2f}  erro=
{erro_total/len(horas):.3f}")

print(f"Prevejo para 6 horas de estudo: {a * 6 + b:.1f}")
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c04_reta

O programa repete mil rodadas. Em cada uma, passa por todos os alunos, calcula os erros e dá um passinho em *a* e em *b*. Ele imprime o estado em rodadas escolhidas:

o computador imprime



```
rodada    1: a=0.86  b=0.25  erro=43.000
rodada    2: a=1.32  b=0.38  erro=12.159
rodada    5: a=1.75  b=0.53  erro=0.558
rodada   10: a=1.81  b=0.58  erro=0.290
rodada   50: a=1.74  b=0.85  erro=0.193
rodada  200: a=1.59  b=1.39  erro=0.077
rodada 1000: a=1.50  b=1.70  erro=0.060
Prevejo para 6 horas de estudo: 10.7
```

Veja o erro: 43 no início, 12 na segunda rodada, 0,56 na quinta, 0,06 na milésima. A queda é rápida no começo e vai ficando lenta. É sempre assim: acertar o grosso é fácil, acertar o fino demora.



Figura 4.3 · O erro cai a cada rodada.

No fim, a máquina chegou a a = 1,50 e b = 1,70. Quem faz a conta exata, com a fórmula dos mínimos quadrados (criada por Legendre e Gauss no começo do século XIX), obtém os mesmos números. A nossa máquina chegou lá devagar, por tentativa; a fórmula chega de uma vez. A vantagem da tentativa é que ela funciona em problemas em que não há fórmula pronta, e esses são quase todos os que importam.

E a previsão? Para 6 horas de estudo, a máquina diz 10,7. Repare: a nota máxima é 10. A máquina não sabe disso! Ela só aprendeu a reta, e a reta continua subindo. É a primeira lição de humildade: a máquina responde, mesmo fora do que viu. Mais adiante voltaremos a isso.



O tamanho do passo importa

O passo é uma escolha nossa, e uma escolha importante. Rodei o mesmo programa com outros passos:

passo	o que aconteceu em 1000 rodadas
0.001	ainda a caminho: a = 1,74 e b = 0,85, erro 0,19
0.02	chegou: a = 1,50 e b = 1,70, erro 0,06
0.1	explodiu: os números cresceram sem parar, até a ordem de 10 elevado a 135
0.2	quebrou de vez: o resultado deixou de ser número (aparece “nan”)

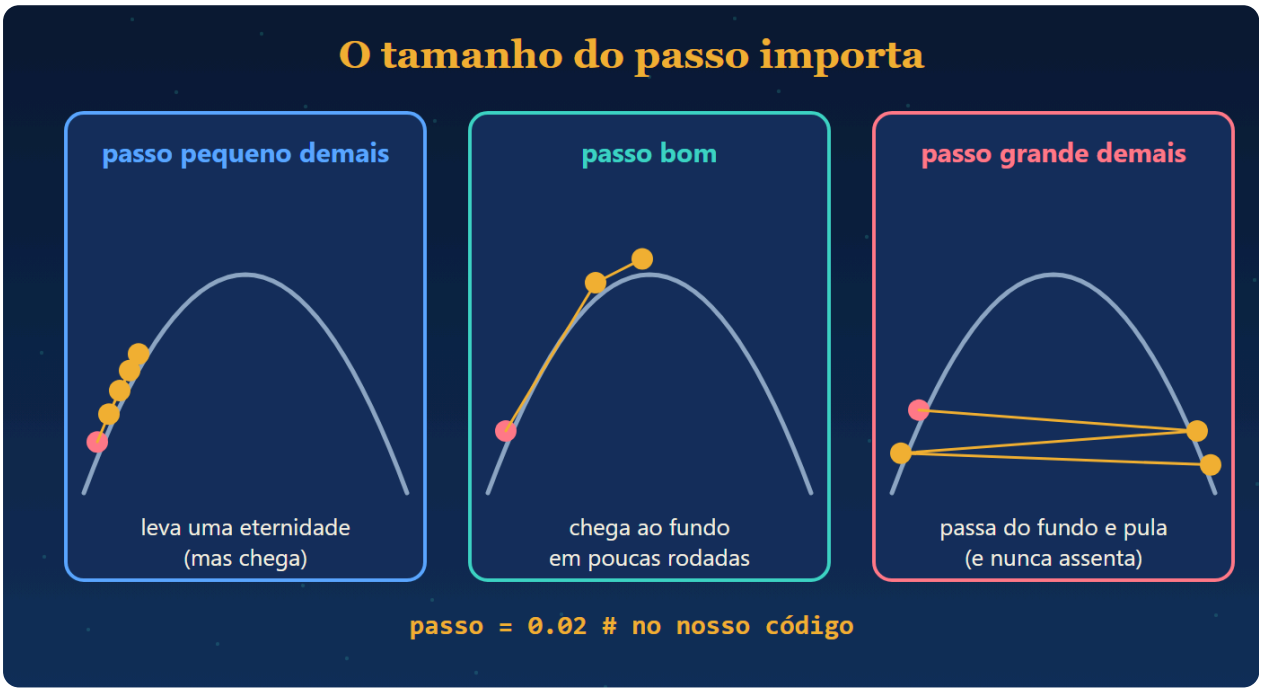


Figura 4.4 · O tamanho do passo importa.

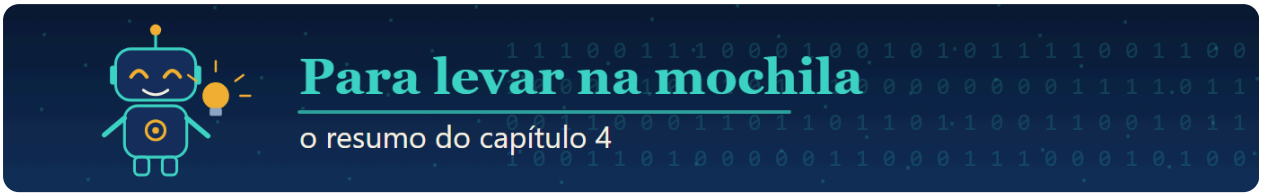
Passo pequeno demais não chega a tempo. Passo grande demais passa do vale, pula para o outro lado, pula de volta e se afasta cada vez mais. Achar o passo certo é parte do ofício.



O que isso tem a ver com a IA de verdade?

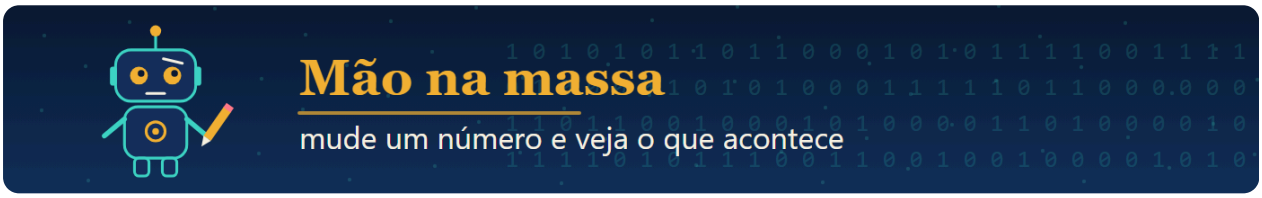
Tudo. O ChatGPT e os outros assistentes têm bilhões de números a ajustar, não dois. Mas o processo é o mesmo: um palpite, o erro, a correção, mil vezes, milhões de vezes. A diferença é de tamanho, não de ideia. Quem entendeu a reta dos cinco alunos entendeu a lógica de tudo.

O que o Bit aprendeu



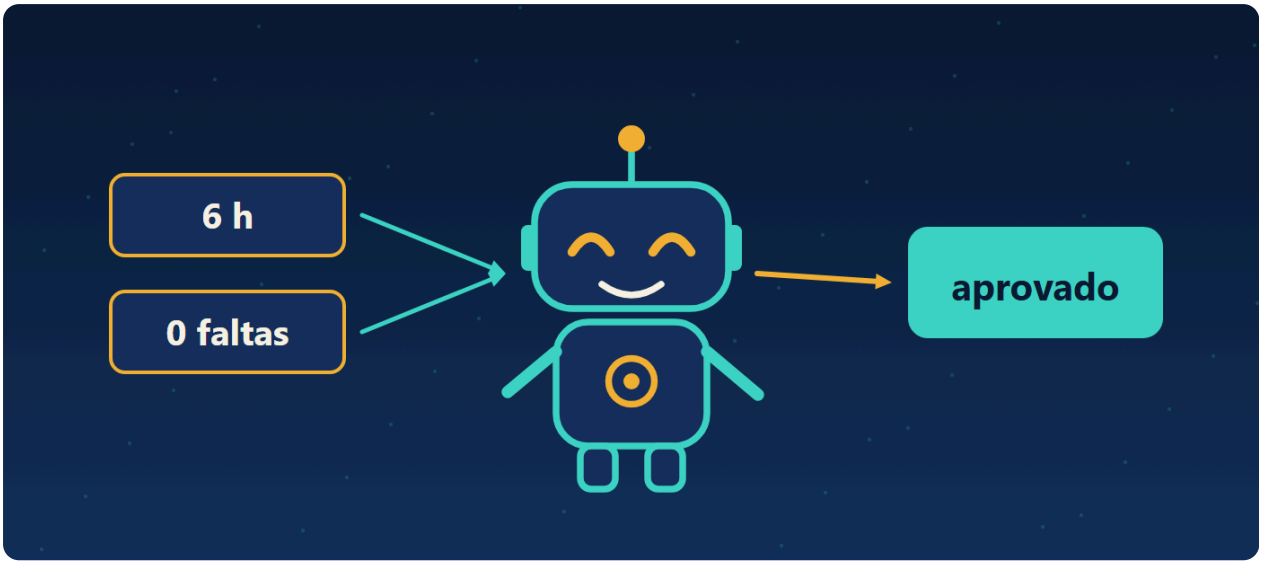
- Aprender é palpitar, medir o erro e corrigir um pouquinho, de novo e de novo.
- A descida do gradiente é um jeito de descobrir para que lado o erro diminui.
- O tamanho do passo precisa ser bem escolhido.
- A máquina responde mesmo fora do que viu, e pode errar sem avisar.

Experimente



Troque `passo = 0.02` por `0.001` e depois por `0.1` e veja os resultados da tabela com os seus próprios olhos. Depois mude as notas para `[2, 4, 6, 8, 10]`: quanto vale o a? E o b? Você consegue prever antes de rodar?

CAPÍTULO 5: O NEURÔNIO



No capítulo anterior a máquina aprendeu uma reta: dava um número, a nota. Agora ela precisa decidir: sim ou não, aprovado ou reprovado, spam ou não spam. Para isso nasceu a peça mais famosa da inteligência artificial, o neurônio artificial.

O neurônio que você tem na cabeça

Um neurônio de verdade é uma célula que recebe sinais de muitas outras. Se a soma dos sinais passa de certo ponto, ele dispara e manda um sinal adiante; se não passa, fica quieto. Em 1943, o neurologista Warren McCulloch e o matemático Walter Pitts propuseram um modelo bem simplificado disso, com números: era o primeiro neurônio artificial.

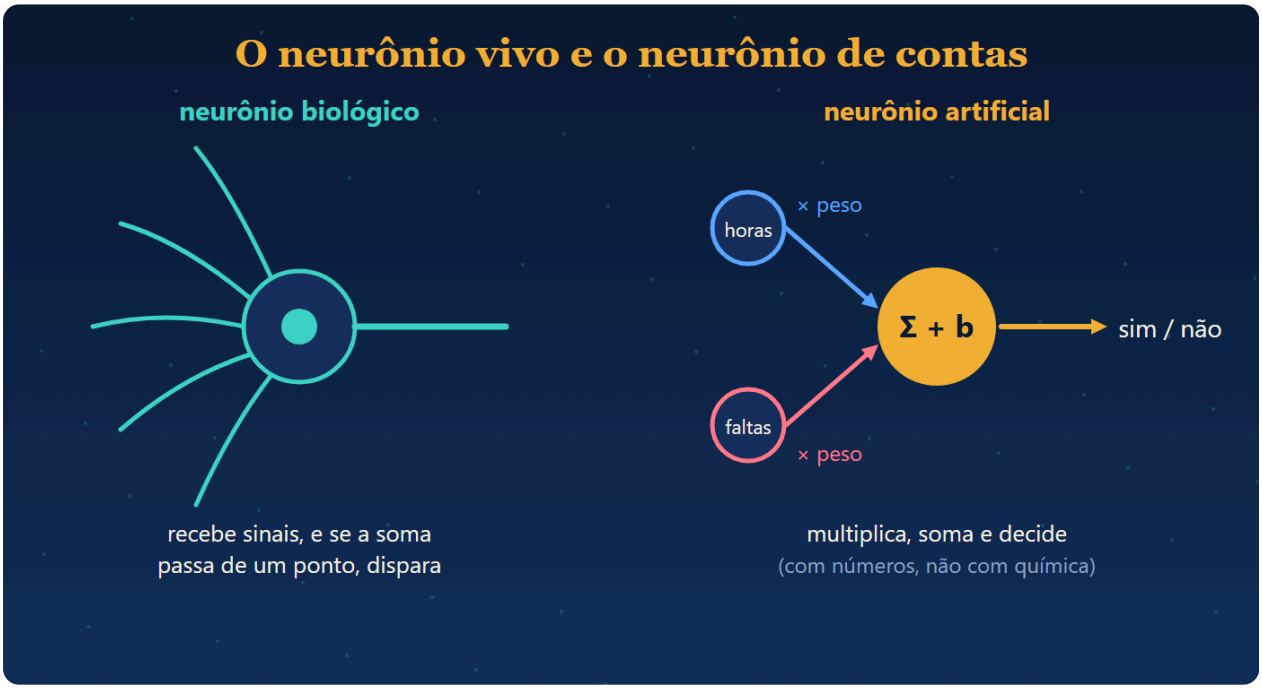


Figura 5.1 · O neurônio vivo e o neurônio de contas.



O neurônio artificial faz só três coisas, e você já sabe todas: multiplica, soma e compara. Cada entrada é multiplicada por um peso, que diz o quanto ela importa. Soma-se tudo, mais um número fixo chamado viés. Se o resultado passa de zero, a resposta é sim; senão, é não.



Um exemplo: aprovado ou reprovado

Voltemos aos alunos. Agora cada um tem duas informações: quantas horas estudou e quantas aulas faltou. Queremos que o neurônio decida se o aluno será aprovado. Intuitivamente, estudar ajuda (peso positivo) e faltar atrapalha (peso negativo). Mas quanto? Nós não sabemos. O neurônio vai descobrir.

Oito alunos, com a resposta certa: aprovados são os que estudaram 6 horas e faltaram 0, 5 e 1, 7 e 2, e 4 e 0. Reprovados são 1 e 5, 2 e 4, 0 e 3, 3 e 6. Na cabeça de qualquer um, existe um limite entre os dois grupos, mas ninguém escreveu a regra.



A regra de aprender do perceptron

Em 1958, o psicólogo americano Frank Rosenblatt apresentou o perceptron, um neurônio desses que aprende, e depois construiu uma máquina para ele. A regra é de uma simplicidade comovente: passe por cada aluno; se o neurônio acertou, não mexa em nada; se errou, empurre os pesos na direção que teria evitado o erro. Quem devia ser aprovado e não foi, ganha mais peso nas entradas que ele tinha; quem não devia e foi, perde.

Rode você mesmo



c05_neuronio.py



```
# Um neurônio (perceptron) aprende a separar aprovados de reprovados.
# Cada aluno: (horas de estudo, faltas) -> 1 aprovado, 0 reprovado
alunos = [((6, 0), 1), ((5, 1), 1), ((7, 2), 1), ((4, 0), 1),
          ((1, 5), 0), ((2, 4), 0), ((0, 3), 0), ((3, 6), 0)]

peso_horas, peso_faltas, vies = 0.0, 0.0, 0.0

def decidir(horas, faltas):
    soma = peso_horas * horas + peso_faltas * faltas + vies
    return 1 if soma > 0 else 0

for rodada in range(1, 21):
    acertos = 0
    for (horas, faltas), certo in alunos:
        erro = certo - decidir(horas, faltas)
        if erro == 0:
            acertos += 1
        peso_horas += erro * horas
        peso_faltas += erro * faltas
        vies += erro
    print(f"rodada {rodada}: {acertos} acertos de {len(alunos)}")
    if acertos == len(alunos):
        break

print("pesos:", peso_horas, peso_faltas, vies)
print("Aluno novo (5 horas, 3 faltas):", "aprovado" if decidir(5, 3) else "reprovado")
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c05_neuronio

o computador imprime

```
rodada 1: 6 acertos de 8
rodada 2: 8 acertos de 8
pesos: 5.0 -5.0 0.0
Aluno novo (5 horas, 3 faltas): aprovado
```

Na primeira rodada o neurônio acertou 6 dos 8 alunos, aprendendo enquanto errava. Na segunda, acertou os oito, e o programa parou. O que ele aprendeu cabe em três números: o peso das horas é 5, o peso das faltas é −5 e o viés é zero.

Leia esses números como uma regra de gente: “cada hora de estudo vale tanto quanto uma falta, só que com o sinal contrário”. Se horas menos faltas for maior que zero, aprovado. Ninguém escreveu essa regra. Foi o neurônio que a achou.

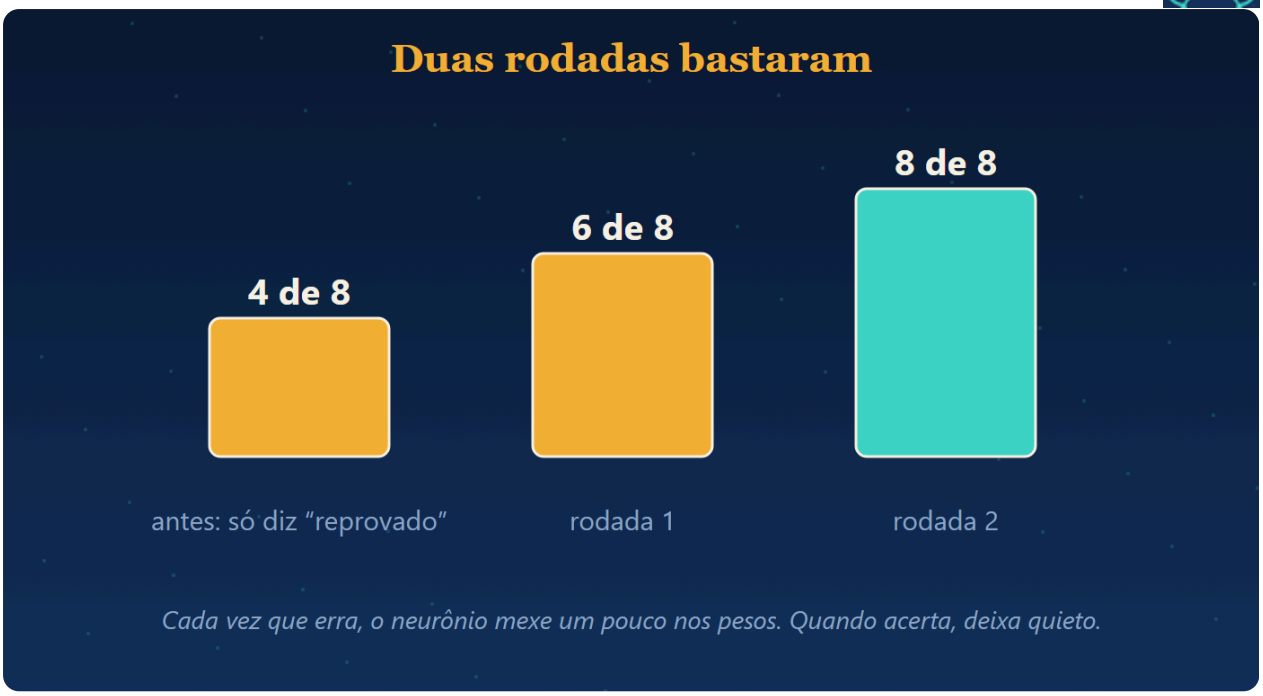


Figura 5.2 · Duas rodadas bastaram.

O aluno novo

Agora o teste de verdade: um aluno que o neurônio nunca viu, com 5 horas de estudo e 3 faltas. A conta é 5 vezes 5, menos 5 vezes 3, mais zero: 25 menos 15, igual a 10. Dez é maior que zero. O neurônio diz: aprovado.

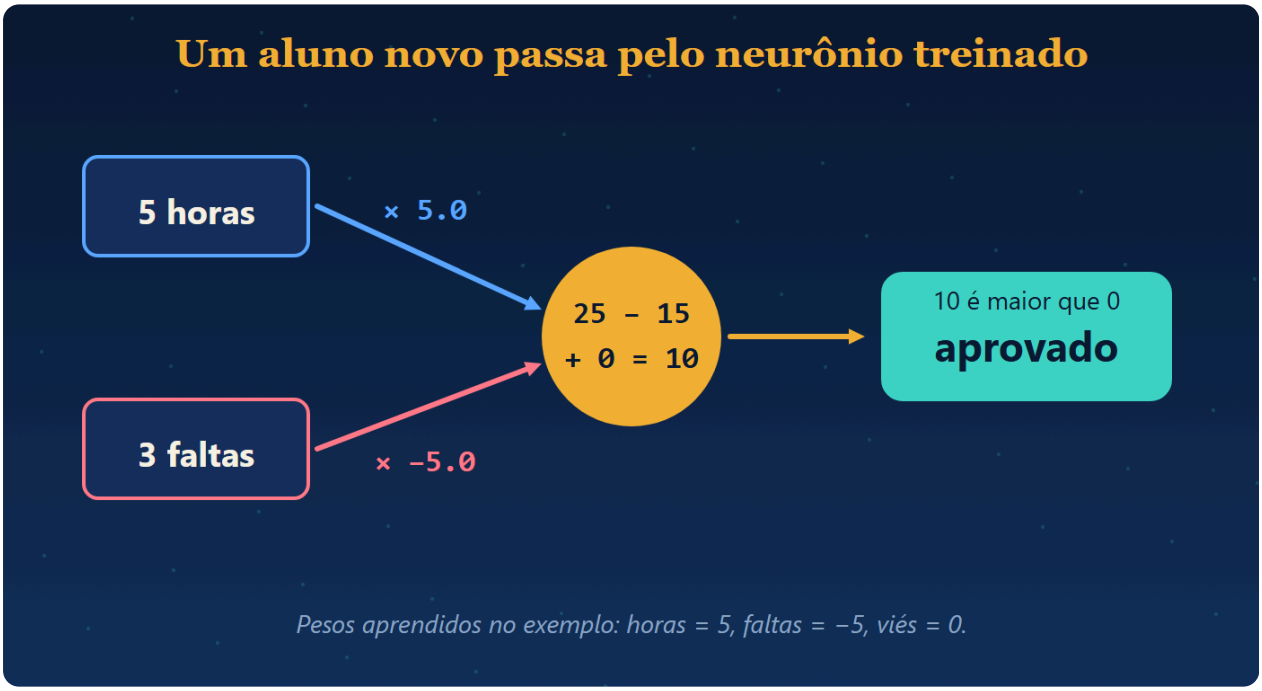


Figura 5.3 · O aluno novo passa pelo neurônio treinado.

É uma boa resposta? Dá para conferir no mapa: marcamos no gráfico cada aluno como um ponto, com as horas na horizontal e as faltas na vertical. O que o neurônio aprendeu foi traçar uma reta que separa os dois grupos. O aluno novo caiu do lado dos aprovados.

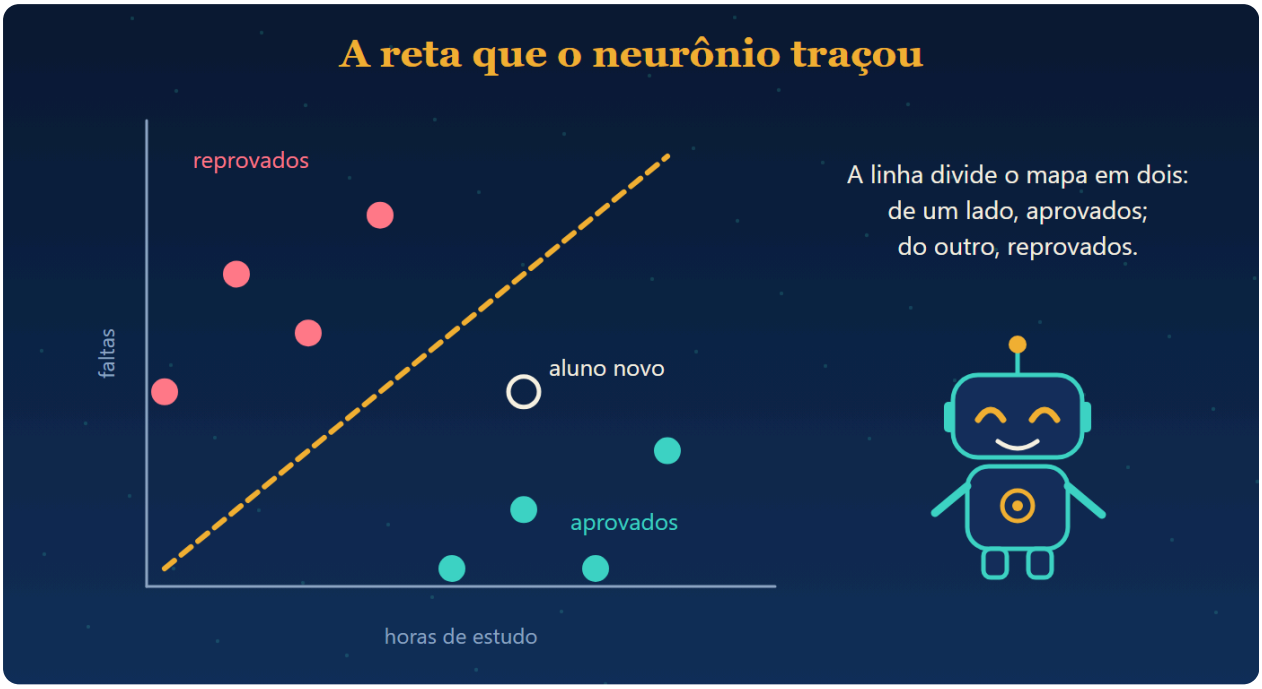


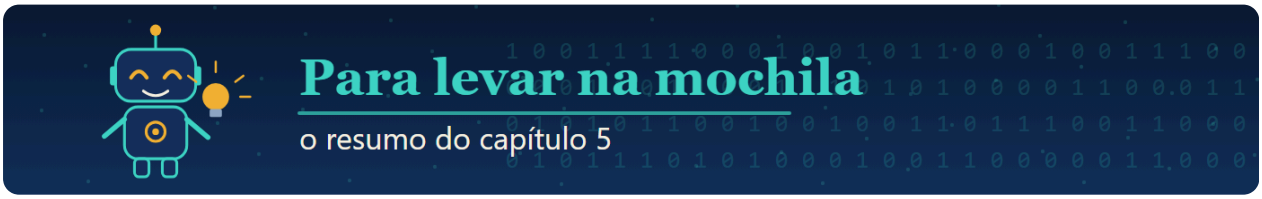
Figura 5.4 · A reta que o neurônio traçou.



Quando a imprensa prometeu demais

O perceptron foi um sucesso tão grande que, em 1958, a imprensa americana chegou a anunciar máquinas que no futuro andariam, falariam e teriam consciência. A promessa não se cumpriu, e voltou como desilusão. Em 1969, Marvin Minsky e Seymour Papert publicaram um livro, *Perceptrons*, mostrando que um neurônio sozinho não resolve problemas em que os grupos não podem ser separados por uma reta. Foi um dos motivos de a área esfriar por anos. O problema clássico apontado por eles é o do próximo capítulo.

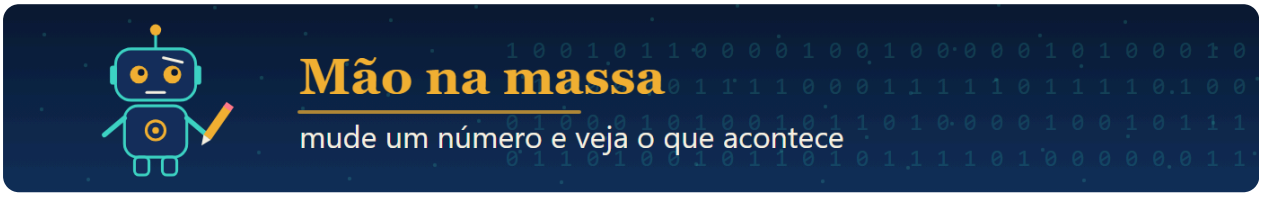
O que o Bit aprendeu





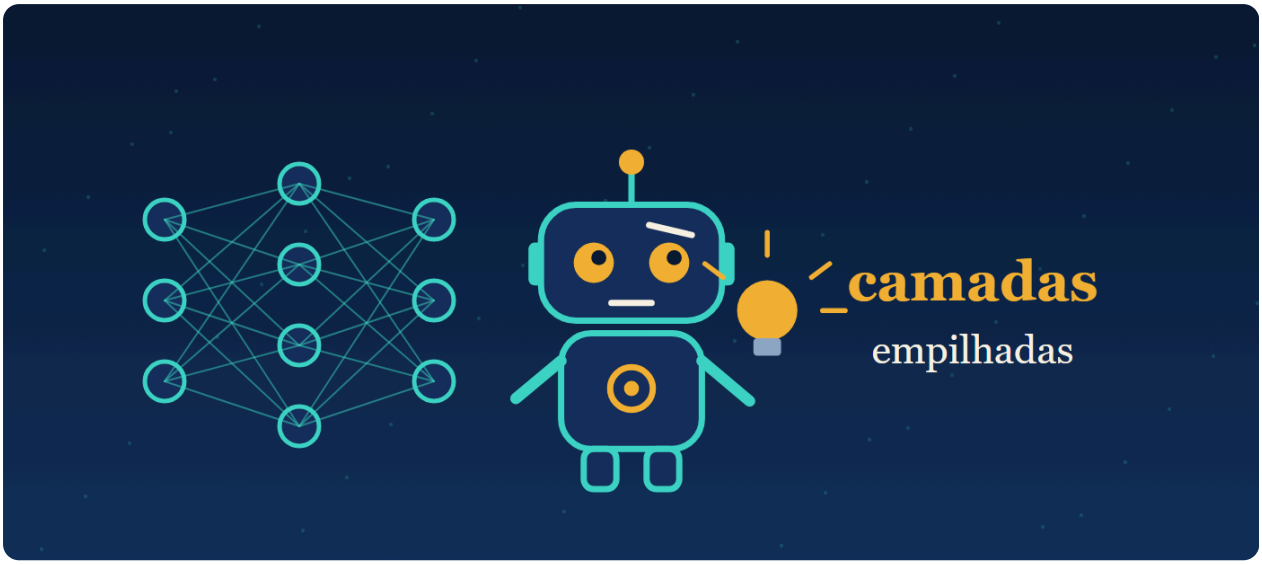
- O neurônio artificial multiplica, soma e compara; os pesos dizem quanto cada entrada importa.
- Ele aprende ajustando os pesos sempre que erra.
- O que ele aprende, no fundo, é uma reta que separa dois grupos.
- Pesos são a memória da máquina: tudo o que ela “sabe” está neles.

Experimente



Troque o último aluno, `((3, 6), 0)`, por `((7, 6), 0)`: alguém que estudou muito, mas faltou muito, e foi reprovado. Rode de novo. Desta vez o neurônio leva três rodadas, não duas, e termina com pesos diferentes: 4 para as horas e −11 para as faltas. Ele aprendeu que faltar pesa muito mais do que estudar. Isso lhe parece razoável? Mude os alunos como quiser e observe como a regra se ajusta.

CAPÍTULO 6: A REDE



Um neurônio traça uma reta. Muita coisa na vida não se resolve com uma reta só. Neste capítulo, o Bit vai ligar vários neurônios em camadas, e a máquina vai ganhar um poder novo: aprender curvas, cantos e combinações que nenhum neurônio sozinho aprenderia.

O problema que parou a área: o XOR

XOR é o "ou exclusivo" da lógica. Recebe dois valores, 0 ou 1, e responde 1 apenas quando os dois são *diferentes*. Se são iguais, responde 0. A tabela é pequena: 0 e 0 dá 0; 0 e 1 dá 1; 1 e 0 dá 1; 1 e 1 dá 0.

Marque as quatro combinações num mapa, e vai ver que nenhuma reta separa os 1 dos 0: os dois pontos de saída 1 ficam em cantos opostos, assim como os dois de saída 0. Um neurônio sozinho só sabe traçar uma reta, então não aprende o XOR. Foi esse exemplo que Minsky e Papert usaram, em 1969, para mostrar o limite do perceptron.

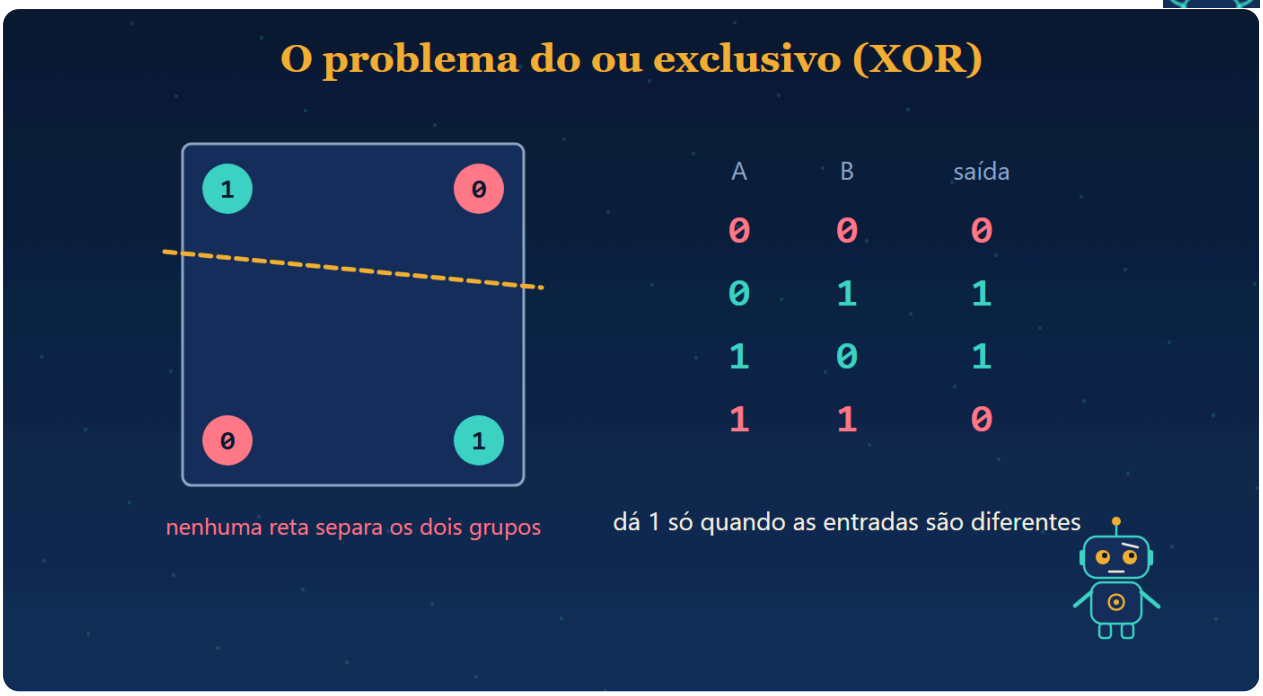


Figura 6.1 · O ou exclusivo (XOR) não se resolve com uma reta.

A saída do impasse: camadas

A solução é conhecida desde cedo: se um neurônio traça uma reta, dois ou três neurônios podem traçar duas ou três retas, e um quarto neurônio pode combinar o que elas disseram. Em vez de uma decisão só, uma camada de neurônios "escondidos" faz perguntas intermediárias, e a camada final junta as respostas. A pergunta difícil quebra-se em perguntas fáceis.

Nosso exemplo usa a rede mais simples possível: duas entradas, uma camada escondida de quatro neurônios e uma saída. Cada linha do desenho é um peso, e há 12 pesos mais os vieses. A rede tem, portanto, muito mais para aprender do que o perceptron.

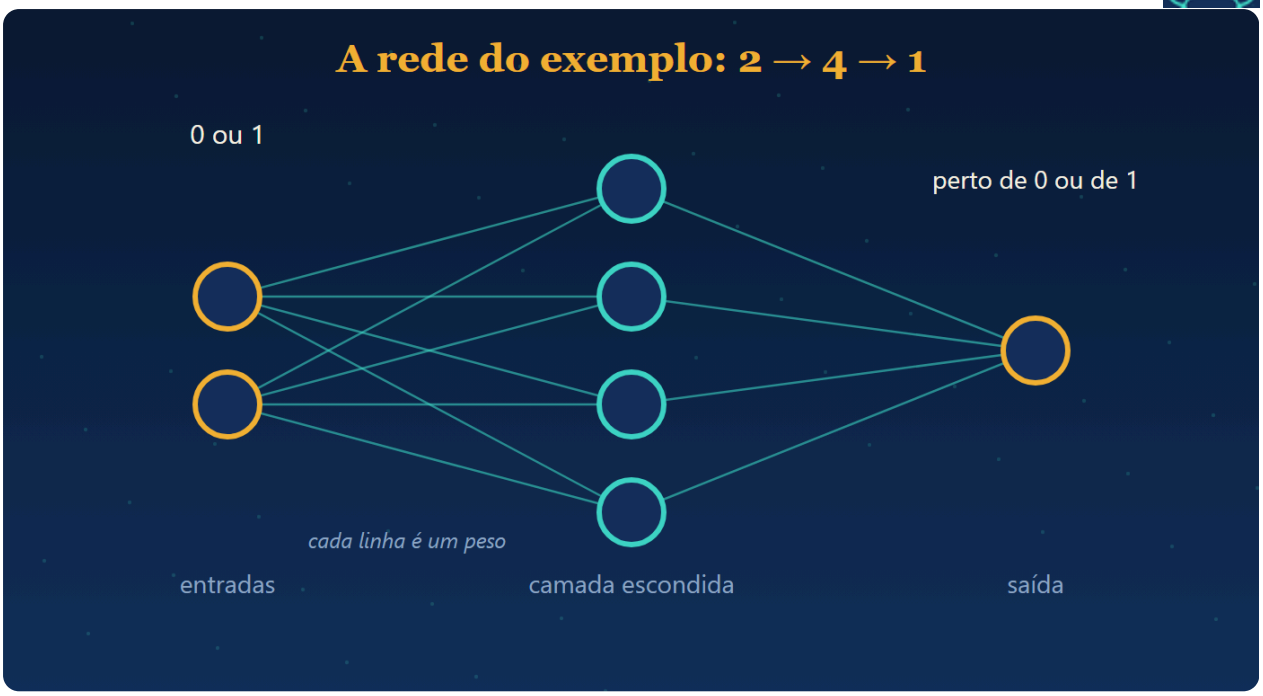


Figura 6.2 · A rede do exemplo: 2, 4 e 1.

Como a rede aprende: a volta do erro

O problema agora é o seguinte: se a resposta da rede está errada, qual dos doze pesos é o culpado? O erro aparece só na saída. Os pesos da primeira camada estão longe dali.

A resposta se chama retropropagação, ou *backpropagation*. É a mesma ideia do capítulo 4, a descida do gradiente, aplicada com um truque: a rede calcula a resposta indo da entrada à saída; depois mede o erro; e então o erro volta, caminhando no sentido contrário, e cada peso descobre o quanto contribuiu para ele. Cada um se corrige na proporção da sua culpa.

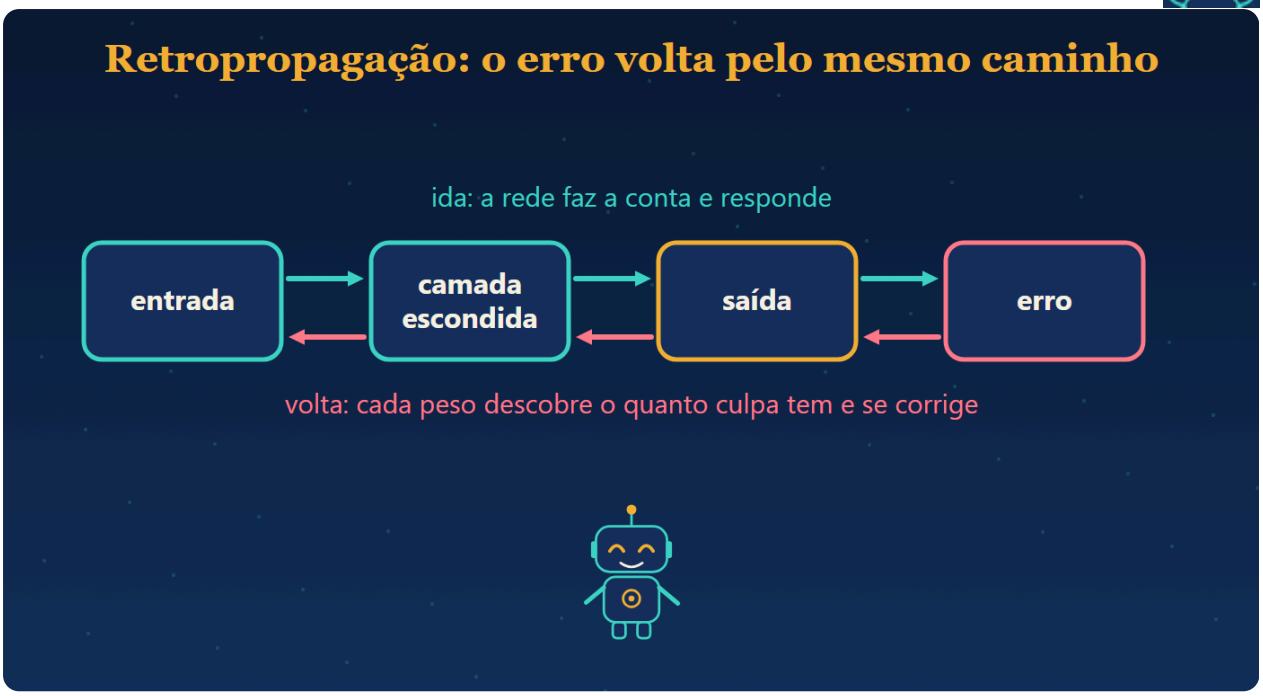


Figura 6.3 · O erro volta pelo mesmo caminho.

Ideias parecidas já circulavam desde os anos 1960 e 1970, mas o método ganhou fama em 1986, num artigo de David Rumelhart, Geoffrey Hinton e Ronald Williams, que mostrou que redes com camadas escondidas aprendiam representações úteis. É por esse ano que a história costuma marcar a volta do interesse nas redes neurais.

Rode você mesmo

Este é o programa mais longo do livro, e o primeiro a usar uma biblioteca de fora, a `numpy`, que faz contas com tabelas de números (as matrizes) com muita rapidez. Para instalar: `pip install numpy`. O sinal `@` multiplica matrizes. O resto são as ideias que você já conhece.



c06_rede_xor.py



```
# Uma rede com uma camada escondida aprende o XOR, que um neurônio sozinho não
aprende.
import numpy as np

entradas = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
alvo = np.array([[0], [1], [1], [0]], dtype=float)

rng = np.random.default_rng(1)
w1 = rng.normal(size=(2, 4)); b1 = np.zeros((1, 4))
w2 = rng.normal(size=(4, 1)); b2 = np.zeros((1, 1))

def sigmoide(x):
    return 1 / (1 + np.exp(-x))

for rodada in range(1, 5001):
    escondida = sigmoide(entradas @ w1 + b1)
    saida = sigmoide(escondida @ w2 + b2)
    erro = saida - alvo
    # volta corrigindo, de trás para frente (retropropagação)
    d_saida = erro * saida * (1 - saida)
    d_escondida = (d_saida @ w2.T) * escondida * (1 - escondida)
    w2 -= 1.0 * escondida.T @ d_saida; b2 -= 1.0 * d_saida.sum(0, keepdims=True)
    w1 -= 1.0 * entradas.T @ d_escondida; b1 -= 1.0 * d_escondida.sum(0,
keepdims=True)
    if rodada in (1, 100, 1000, 5000):
        print(f"rodada {rodada:4d}: erro médio = {np.mean(erro ** 2):.4f}")

print("Resposta da rede:")
for e, s in zip(entradas, saida):
    print(e.astype(int), "->", round(float(s[0]), 2))
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c06_rede_xor

o computador imprime

```
rodada    1: erro médio = 0.2749
rodada  100: erro médio = 0.2453
rodada 1000: erro médio = 0.0023
rodada 5000: erro médio = 0.0002
Resposta da rede:
[0 0] -> 0.01
[0 1] -> 0.98
[1 0] -> 0.99
[1 1] -> 0.02
```



Olhe o erro. Nas primeiras cem rodadas, quase nada acontece: caiu só de 0,27 para 0,25. Até as mil rodadas, despencou para 0,002. Ao fim, 0,0002. É um comportamento comum: durante um tempo a rede parece perdida e, de repente, "pega". As respostas finais são quase perfeitas: 0,01 para 0 e 0; 0,98 e 0,99 para as combinações diferentes; 0,02 para 1 e 1.



Figura 6.4 · Por um tempo nada acontece; de repente, a rede "pega".

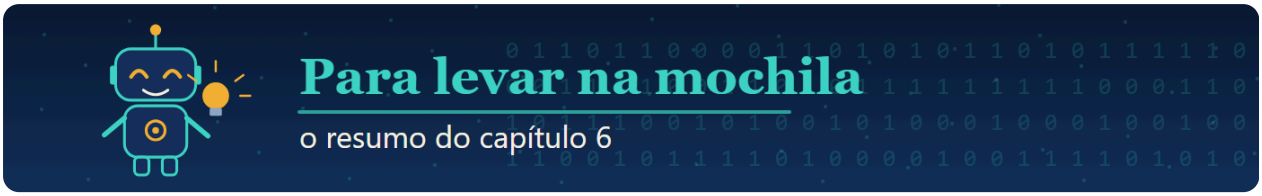
Uma observação honesta: o resultado depende da sorte inicial dos pesos, que vêm de números aleatórios (por isso a linha `default_rng(1)`, que fixa a semente do sorteio). Com outra semente os pesos começam em outros lugares, e em redes maiores isso pode fazer a rede demorar mais, ou até empacar. Neste exemplo pequeno, testei as sementes 1, 2 e 3 e todas aprenderam. Na prática, quem treina redes grandes roda várias vezes e fica com a melhor.



Por que camadas importam

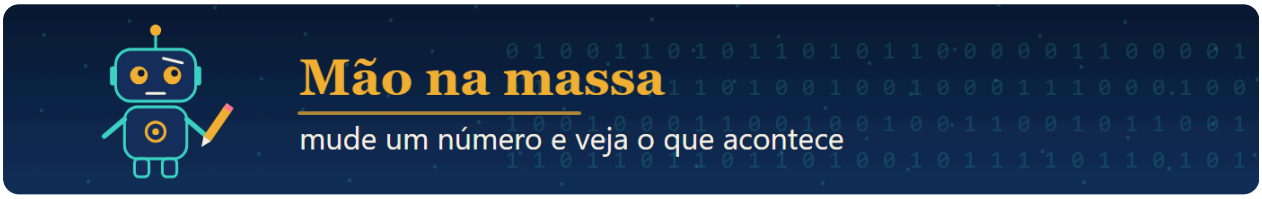
Cada camada aprende a ver o problema de um jeito mais abstrato. Numa rede que reconhece rostos, por exemplo, as primeiras camadas costumam reagir a riscos e bordas, as do meio a olhos e narizes, e as últimas a rostos inteiros. Ninguém programou essa hierarquia. Ela nasceu do treino. É por isso que se chama aprendizado "profundo" (*deep learning*): profundo no sentido de ter muitas camadas.

O que o Bit aprendeu



- Um neurônio só traça retas; vários, em camadas, resolvem problemas que não são retos.
- A retropropagação leva o erro de volta e corrige cada peso na proporção da culpa.
- No começo, a rede pode parecer parada; depois ela pega.
- O resultado depende de uma sorte inicial, e por isso se treina mais de uma vez.

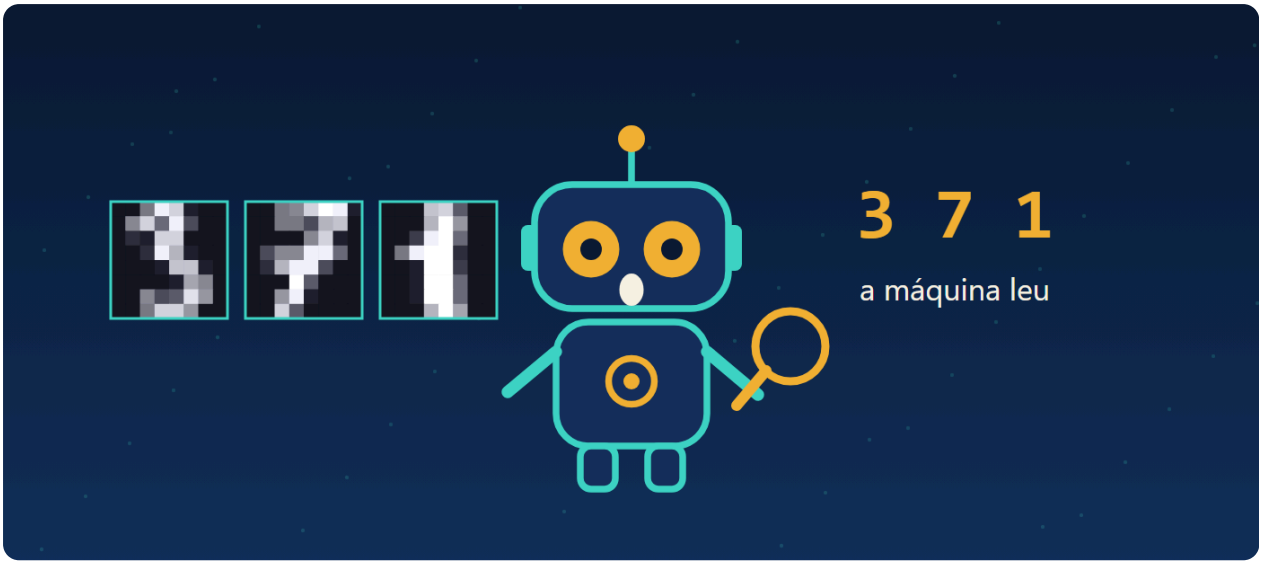
Experimente



Troque `default_rng(1)` por `default_rng(2)` e por `default_rng(3)` e olhe o erro final. Depois reduza a camada escondida de 4 para 2 neurônios: troque o 4 por 2 nas linhas de `w1`, `b1` e `w2`. Eu testei: com dois neurônios escondidos a rede também aprende o XOR, e o erro final fica em 0,0003. Dois bastam, mas com quatro a rede ficou um pouco mais folgada. E com um só? Também testei: não aprende. A rede responde 0,02 para o 0 e o 0 e 0,67 para todo o resto, e o erro fica preso em 0,17. Um neurônio escondido é pouco: é o mesmo impasse do capítulo 5. Tente e veja você mesmo.



CAPÍTULO 7: ENSINAR A MÁQUINA A VER



Já vimos o neurônio e a rede. Agora vamos pô-los para trabalhar numa tarefa de verdade, que até pouco tempo era coisa de ficção científica: reconhecer um número escrito à mão. O Bit vai olhar uma imagem e dizer se é um 0, um 1, um 2, até um 9.

As imagens do exercício

Usaremos uma coleção pequena e famosa que vem junto com o scikit-learn, uma biblioteca gratuita de aprendizado de máquina para Python. São 1.797 imagens de dígitos escritos por pessoas, cada uma com 8 por 8 pontos, de uma coleção pública conhecida como UCI. Cada ponto é um número de 0 (escuro) a 16 (claro). Lembra do capítulo 3? Uma imagem é uma grade de números. Aqui são 64 deles por imagem.

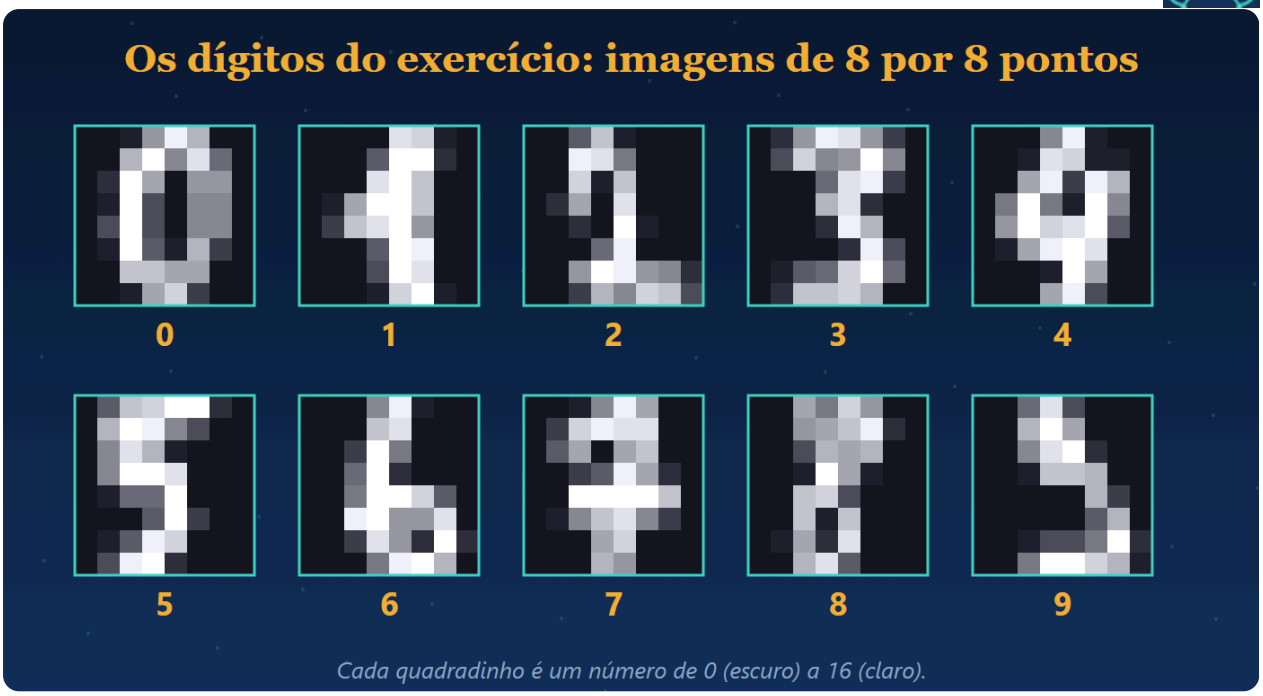


Figura 7.1 · Os dígitos do exercício: cada um é uma grade de 8 por 8.

Veja como elas são tosquinhas: 64 pontos para desenhar um número. Mesmo assim, uma pessoa lê a maioria sem dificuldade. A pergunta é se a máquina consegue.



Treino e teste: a regra de ouro

Há uma regra de ouro em aprendizado de máquina: nunca avalie a máquina com os mesmos exemplos que ela usou para aprender. É como a prova da escola. Se o professor repetir na prova exatamente as questões que resolveu em aula, a nota diz só quem decorou.

Por isso separamos as imagens em dois grupos: o de treino, com 75%, para a rede aprender, e o de teste, com 25%, que ela nunca viu, para medir se aprendeu de verdade. O scikit-learn tem uma função para isso, a `train_test_split`.

Rode você mesmo

O programa a seguir faz tudo: carrega as imagens, separa treino e teste, cria uma rede neural com uma camada escondida de 32 neurônios, treina e mede. No fim, mostra uma imagem do teste e o que a rede acha que é. Para instalar a biblioteca: `pip install scikit-learn`.





c07_digitos.py

```
# A máquina aprende a ler dígitos escritos à mão (imagens 8x8 que vêm com o scikit-
learn).
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

digitos = load_digits()
print("imagens:", len(digitos.images), "tamanho de cada uma:",
digitos.images[0].shape)

x_treino, x_teste, y_treino, y_teste = train_test_split(
    digitos.data, digitos.target, test_size=0.25, random_state=0)

rede = MLPClassifier(hidden_layer_sizes=(32,), max_iter=500, random_state=0)
rede.fit(x_treino, y_treino)
print(f"acerto no treino: {rede.score(x_treino, y_treino):.1%}")
print(f"acerto no teste (imagens nunca vistas): {rede.score(x_teste, y_teste):.1%}")

# Uma imagem do teste, desenhada com texto
imagem = x_teste[0].reshape(8, 8)
for linha in imagem:
    print("".join("#" if p > 8 else "." for p in linha))
print("A rede diz que é o número", rede.predict([x_teste[0]])[0], "- o certo é",
y_teste[0])
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c07_digitos

o computador imprime

```
imagens: 1797 tamanho de cada uma: (8, 8)
acerto no treino: 100.0%
acerto no teste (imagens nunca vistas): 95.8%
..###...
..####..
..#..#..
....##..
....#...
...##...
..####..
..##.##.
A rede diz que é o número 2 - o certo é 2
```



Se aparecer um aviso chamado ConvergenceWarning, não se assuste: significa só que a rede ainda ia melhorar um pouco quando atingiu o limite de 500 rodadas que demos. Para o exemplo, não faz diferença.

No treino a rede acertou 100% (decorou tudo). No teste, imagens que nunca tinha visto, acertou 95,8%. Esse é o número que importa. A imagem do final, desenhada com cerquilhas, é um 2 meio torto, e a rede disse "2". Acertou.

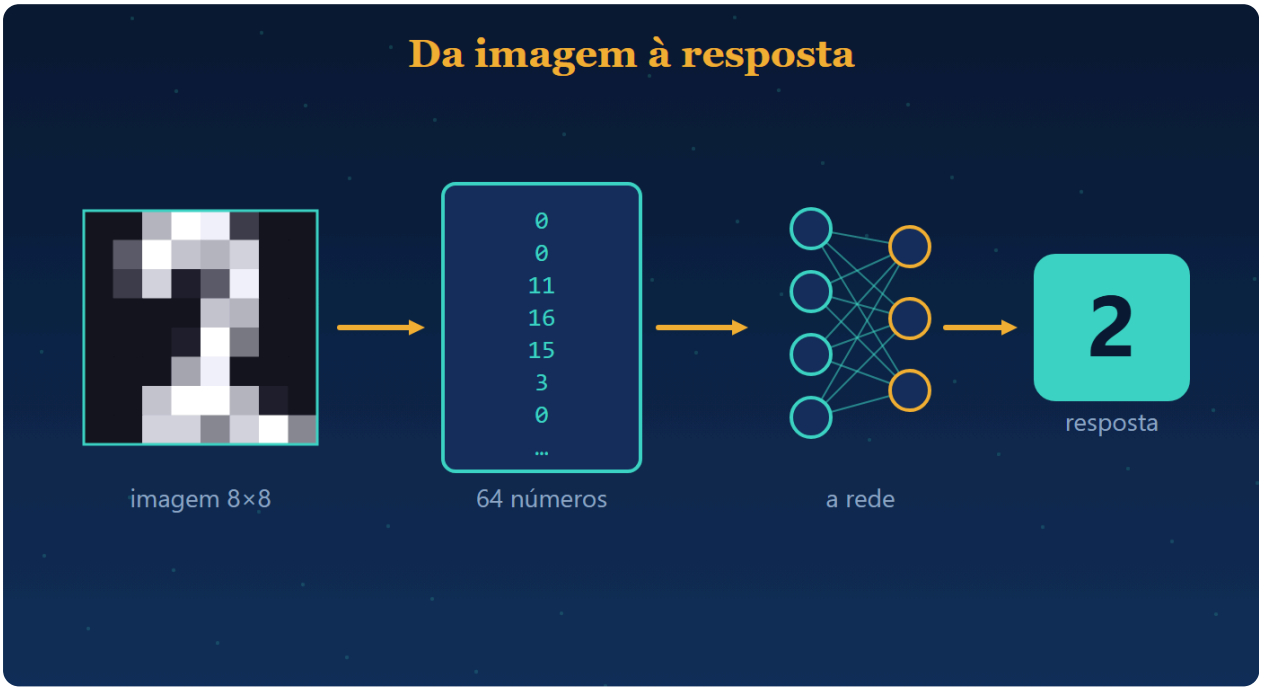


Figura 7.2 · Da imagem à resposta.

O que a rede errou

Dos 450 dígitos do teste, a rede errou 19. A figura abaixo mostra seis deles. Alguns são honestamente difíceis: um 5 que parece 9, um 9 que parece 7. Até uma pessoa hesitaria. Outros nos fazem coçar a cabeça.

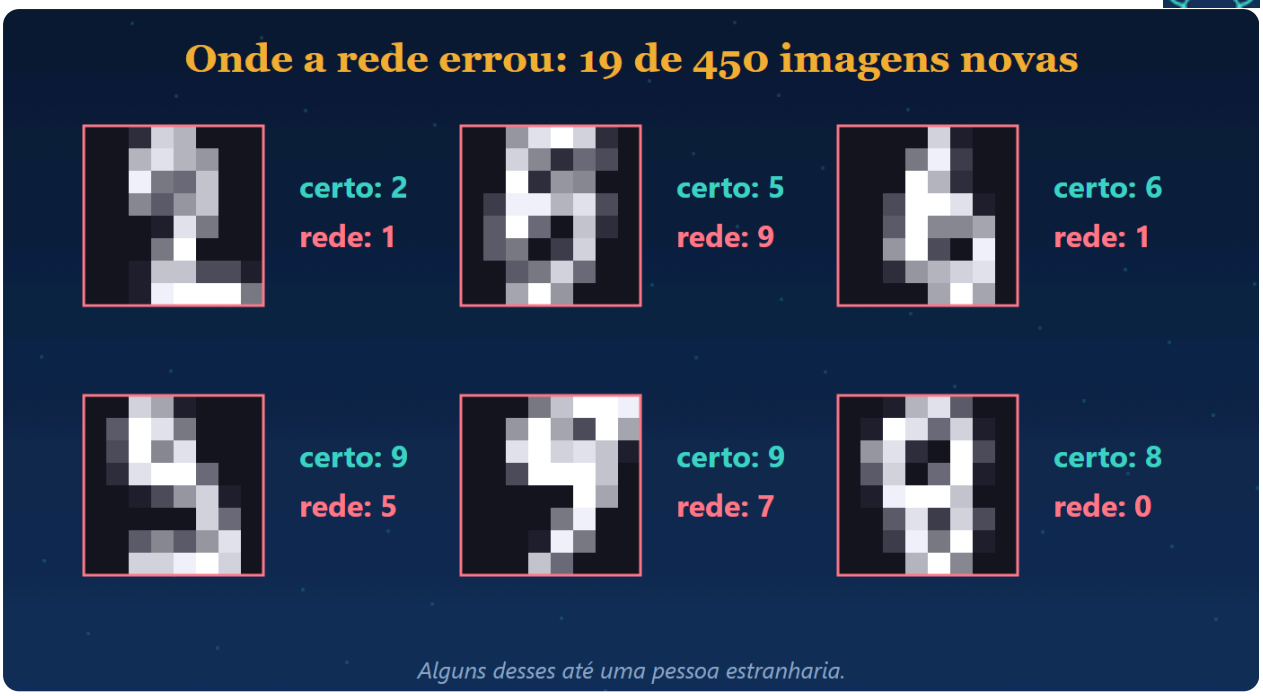


Figura 7.3 · Onde a rede errou.

Aqui está uma lição que vale para toda a inteligência artificial: ela erra de um jeito diferente do nosso. Uma pessoa erra por cansaço, pressa, desatenção. A máquina erra por razões que às vezes não conseguimos explicar. Por isso, quando uma IA decide algo importante, a pergunta não é só "quanto ela acerta?", é também "quando e como ela erra?".

Decorar não é aprender

Vimos que a rede acertou 100% no treino. Isso pode enganar. Se a rede tem neurônios sobrando e poucos exemplos, ela pode decorar os exemplos em vez de aprender o que é um 3. A esse problema dá-se o nome de sobreajuste, em inglês *overfitting*.

Para ver isso acontecer, rodei o mesmo exercício dando à rede poucos, depois mais, depois quase todos os exemplos de treino:



c07b_decorar.py



```
# Decorar não é aprender: com poucos exemplos, a rede vai bem no treino e mal no
teste.
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

d = load_digits()
x_treino, x_teste, y_treino, y_teste = train_test_split(d.data, d.target,
test_size=0.5, random_state=0)
for quantos in (20, 100, 900):
    rede = MLPClassifier(hidden_layer_sizes=(64,), max_iter=2000, random_state=0)
    rede.fit(x_treino[:quantos], y_treino[:quantos])
    print(f"{quantos:4d} exemplos: treino {rede.score(x_treino[:quantos],
y_treino[:quantos]):.0%}, teste {rede.score(x_teste, y_teste):.0%}")
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c07b_decorar

o computador imprime

```
  20 exemplos: treino 100%, teste 44%
 100 exemplos: treino 100%, teste 82%
 900 exemplos: treino 100%, teste 96%
```

Com 20 exemplos, a rede acertou 100% do treino e só 44% do teste: decorou. Com 100, foi a 82%. Com 900, a 96%. Em todos os casos, o treino fica em 100%. Quem olhasse só para ele jamais perceberia a diferença. É por isso que o teste existe.

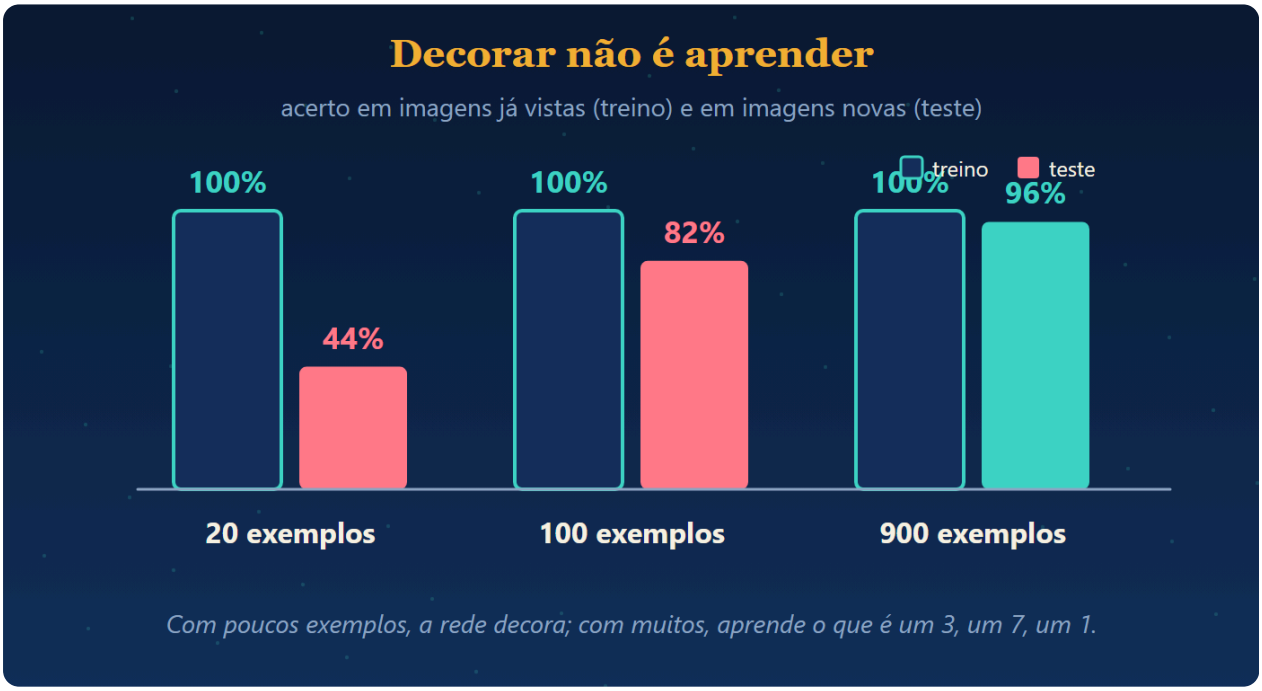


Figura 7.4 · Decorar não é aprender.

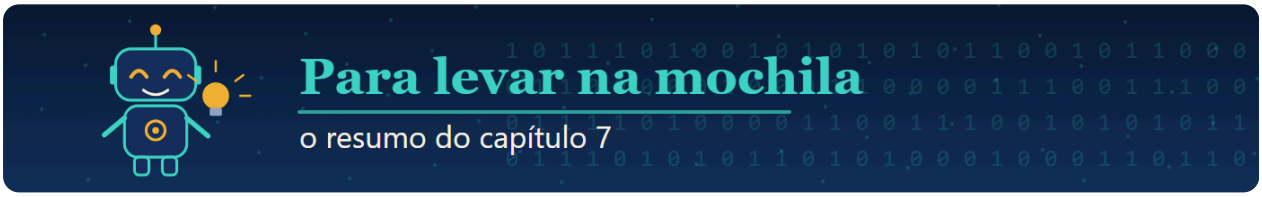


E o mundo real?

As imagens que usamos são minúsculas e comportadas. A leitura de dígitos manuscritos foi um dos primeiros sucessos práticos das redes neurais: no fim dos anos 1980, Yann LeCun e seus colegas, nos laboratórios Bell, treinaram uma rede para ler os números dos códigos postais escritos à mão em cartas americanas. Em 2012, uma rede chamada AlexNet venceu um grande concurso de reconhecimento de imagens com folga, e a maré virou de vez para o aprendizado profundo.

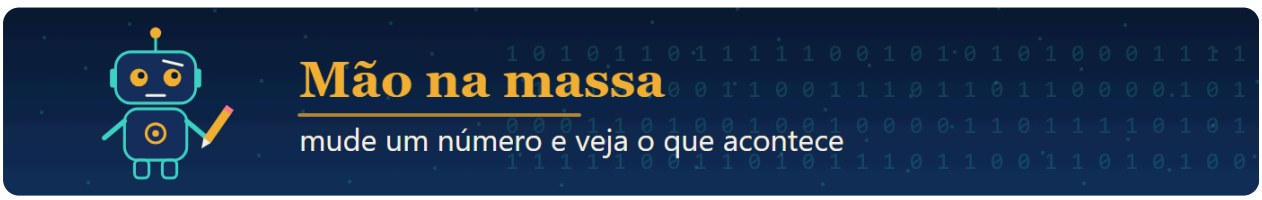
O que muda do nosso exemplo para esses sistemas é a escala: imagens maiores, milhões de exemplos, camadas especializadas em imagens. A ideia, porém, é a mesma: pontos viram números, a rede aprende os pesos, e o teste diz se ela aprendeu ou decorou.

O que o Bit aprendeu



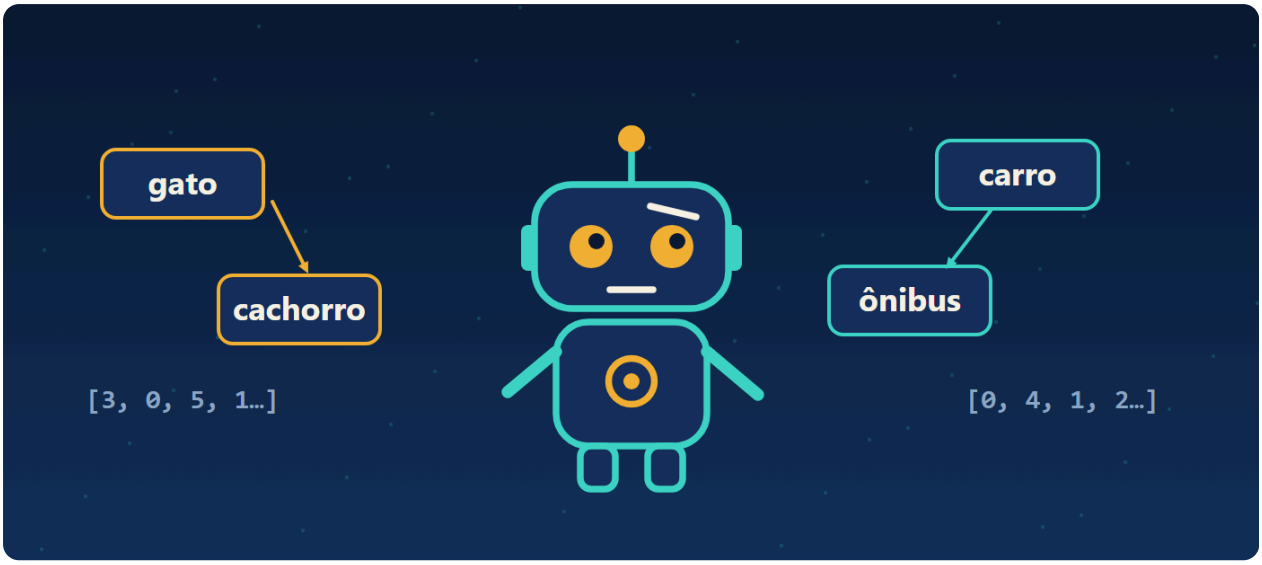
- Uma imagem é uma grade de números, e a rede aprende com milhares delas.
- Sempre se avalia com exemplos que a máquina nunca viu.
- Com poucos exemplos a máquina decora; com muitos, aprende.
- A máquina erra de um jeito diferente do nosso, e vale conhecer esse jeito.

Experimente



Em `c07_digitos.py`, troque `x_teste[0]` por `x_teste[1]`, `x_teste[2]` e assim por diante, e veja se a rede acerta cada um. Depois troque `hidden_layer_sizes=(32,)` por `(4,)`: uma camada escondida minúscula. O acerto no teste cai? Eu testei: vai de 95,8% para 89,3%. E as imagens seguintes do teste (de 1 a 5)? A rede acerta as cinco: 8, 2, 6, 6 e 7.

CAPÍTULO 8: PALAVRAS VIRAM NÚMEROS



Até aqui, tudo o que a máquina aprendeu era numérico por natureza: horas, faltas, pixels. Agora vem o salto que leva ao chat: como fazer uma máquina entender palavras? No capítulo 3 vimos que cada letra tem um número, mas isso só diz como guardar. Não diz o que a palavra *significa*. “Gato” é uma sequência de quatro números de letras, e “cachorro” outra, sem nenhuma relação entre elas. Para a máquina, gato e cachorro seriam tão distantes quanto gato e geladeira.



Uma ideia de 1957

O linguista inglês John Rupert Firth escreveu, em 1957, uma frase que se tornou lema da área: conhece-se uma palavra pela companhia que ela tem. Pense em “gato” e “cachorro”. Ambos bebem, comem, dormem, ficam em casa, são bichos de estimação. Já “carro” e “ônibus” andam, param, rodam na rua. Quem aparece perto de quem mostra o que as palavras são.

Foi essa ideia, bem antiga, que levou ao método moderno: em vez de dar à palavra um número arbitrário, damos a ela uma lista de números que resume as suas companhias. Palavras com companhias parecidas ganham listas parecidas. Chamamos essa lista de vetor da palavra, ou, na linguagem técnica, de *embedding*.

Contar quem anda com quem

Vamos fazer isso numa escala de brinquedo. Escrevi nove frases curtas sobre gatos, cachorros, carros e ônibus. Para cada palavra, o programa conta quantas vezes ela aparece ao lado de cada uma das outras. Essa contagem é o vetor da palavra.



Figura 8.1 · Quantas vezes cada palavra aparece ao lado de outra.

Gato e cachorro têm linhas quase idênticas: ambos bebem, comem e dormem no sofá. Carro e ônibus também. Mas as linhas dos animais e as dos veículos não se parecem. Nenhuma palavra foi explicada à máquina; ela só contou.



Medir a semelhança

Como saber o quanto duas listas de números se parecem? Existe uma medida chamada cosseno, que dá 1 para listas que apontam na mesma direção e 0 para listas sem nada em comum. Você pode imaginar cada vetor como uma seta; o cosseno mede o ângulo entre duas setas.

Rode você mesmo



c08_palavras.py

```
# Palavras viram números: cada palavra ganha um vetor segundo as companhias que tem.
import numpy as np

frases = [
    "o gato bebe leite", "o cachorro bebe agua", "o gato come peixe",
    "o cachorro come osso", "o carro anda na rua", "o onibus anda na rua",
    "o gato dorme no sofa", "o cachorro dorme no sofa", "o carro para na rua",
]
vocab = sorted({p for f in frases for p in f.split()})
pos = {p: i for i, p in enumerate(vocab)}

# conta quantas vezes cada palavra aparece ao lado de cada outra
cont = np.zeros((len(vocab), len(vocab)))
for f in frases:
    ps = f.split()
    for i, p in enumerate(ps):
        for j, q in enumerate(ps):
            if i != j:
                cont[pos[p], pos[q]] += 1

def vetor(p):
    return cont[pos[p]]

def parecido(a, b):
    va, vb = vetor(a), vetor(b)
    return float(va @ vb / (np.linalg.norm(va) * np.linalg.norm(vb)))

for a, b in [("gato", "cachorro"), ("gato", "carro"), ("carro", "onibus"),
("cachorro", "onibus")]:
    print(f"{a:9s} x {b:9s}: {parecido(a, b):.2f}")
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c08_palavras

o computador imprime

```
gato      x cachorro : 0.88
gato      x carro    : 0.40
carro     x onibus   : 0.94
cachorro  x onibus   : 0.38
```

O resultado diz o que esperávamos. Gato e cachorro: 0,88. Carro e ônibus: 0,94. Gato e carro: 0,40. Cachorro e ônibus: 0,38. A máquina nunca soube o que é um gato. Mas descobriu, só pela vizinhança das palavras, que gato é mais parecido com cachorro do que com carro.



Figura 8.2 · Quão parecidas são duas palavras?

Um mapa de palavras

Dá para desenhar isso num mapa. Os vetores têm muitas dimensões, mas existe uma técnica (a decomposição em valores singulares, SVD) que os resume em duas, para caber no papel. O resultado põe cada palavra num ponto, e palavras parecidas ficam juntas. No nosso mapa, como o texto é minúsculo, gato e cachorro caem no mesmo ponto: companhias iguais, lugar igual.

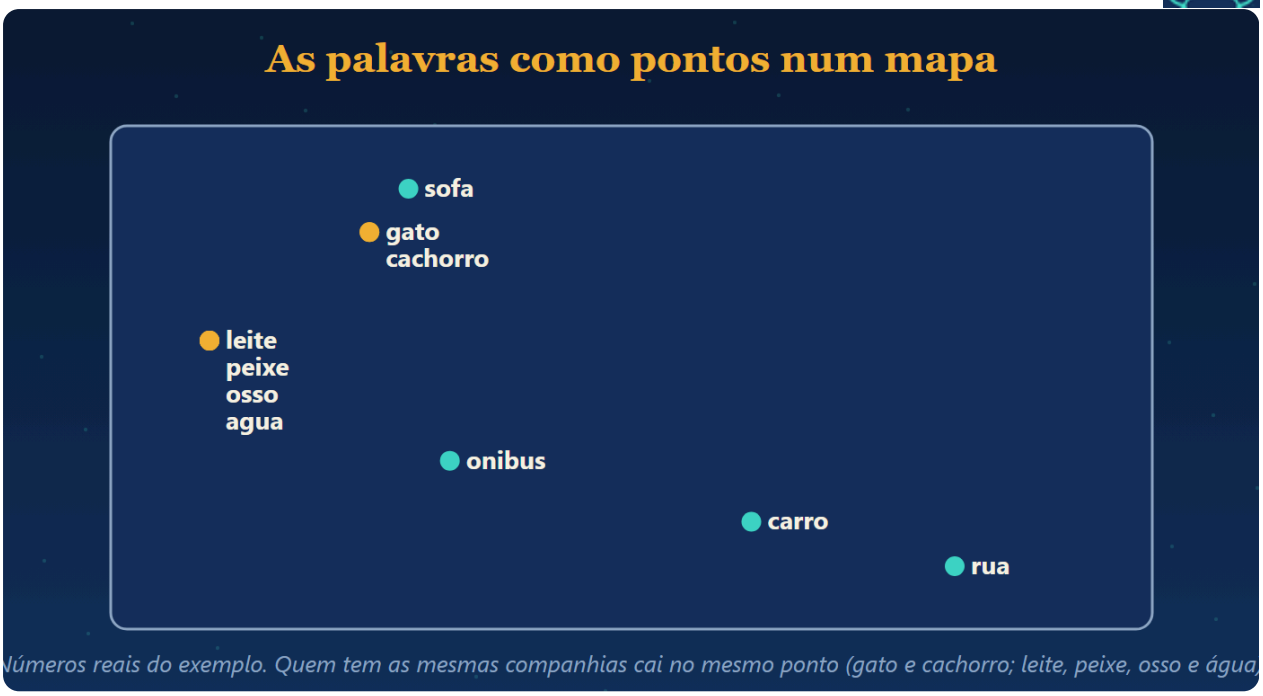


Figura 8.3 · Quem tem as mesmas companhias cai no mesmo lugar.

Em escala de verdade

Em 2013, uma equipe do Google liderada por Tomas Mikolov publicou um método, o *word2vec*, que aprendia vetores de palavras lendo bilhões de palavras. Os vetores eram bem mais ricos do que os nossos, e mostraram uma propriedade que encantou os pesquisadores: dava para fazer conta com palavras. O exemplo clássico: o vetor de "rei", menos o de "homem", mais o de "mulher", ficava perto do vetor de "rainha".

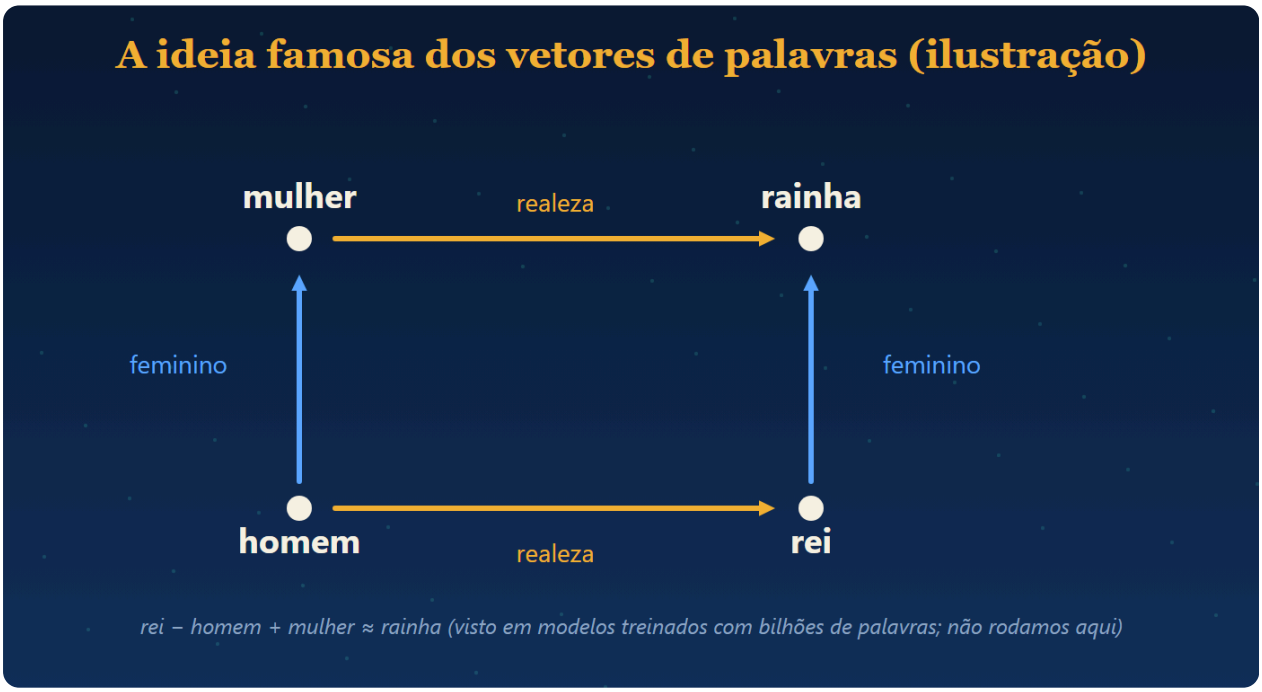


Figura 8.4 · A ideia famosa dos vetores de palavras (ilustração).



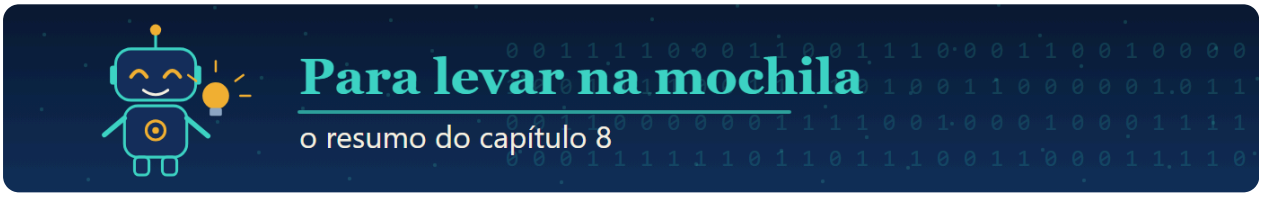
Aviso de honestidade: nosso exemplo de brinquedo, com nove frases, não faz isso. A figura acima é uma ilustração do que acontece em modelos grandes, treinados com muito texto. Não rodamos isso aqui, e não espere esse resultado de nenhum código curto.



O que os vetores aprendem, e o que eles herdam

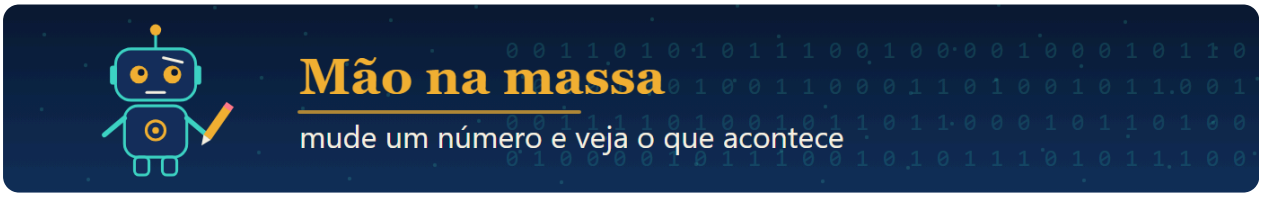
Aprender com textos de gente traz um efeito colateral: os vetores aprendem também os preconceitos dos textos. Se, nos textos, certas profissões aparecem mais ao lado de certos gêneros, os vetores registram isso, e uma máquina que os use pode repetir o estereótipo. Isso foi mostrado em pesquisas e é um dos cuidados do último capítulo.

O que o Bit aprendeu



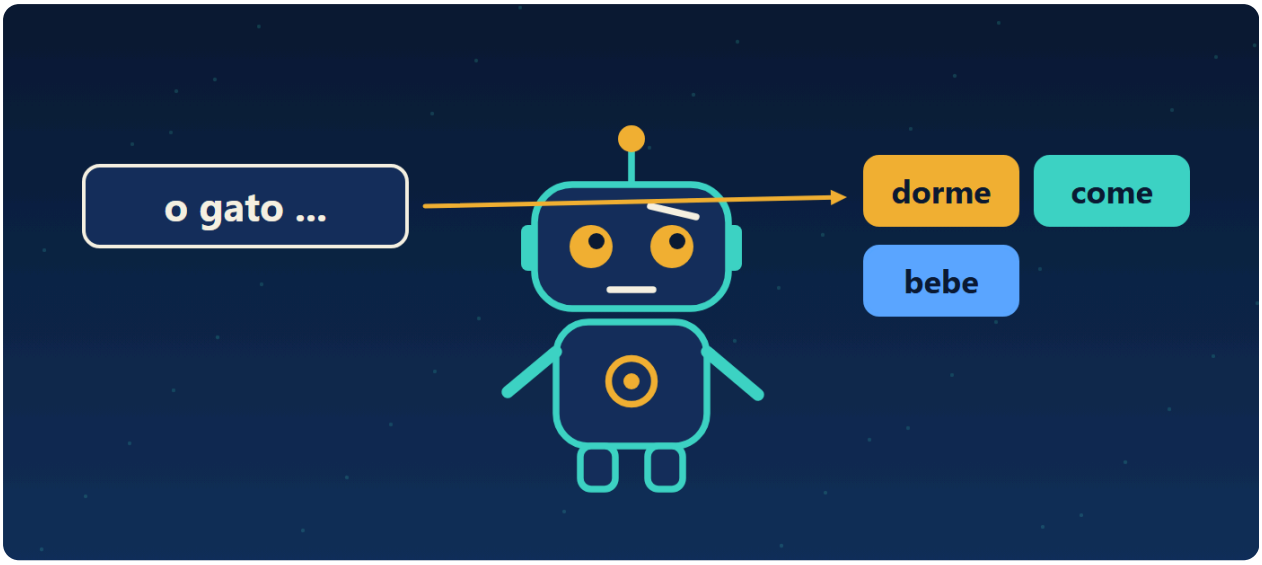
- Cada palavra ganha uma lista de números que resume as suas companhias.
- Palavras com companhias parecidas têm vetores parecidos.
- A máquina não sabe o que é um gato: sabe com quem o gato anda.
- Vetores herdam o que existe nos textos, inclusive os preconceitos.

Experimente



Acrescente duas frases à lista `frases`, por exemplo `"o gato anda no sofa"` e `"o carro dorme na rua"`. Rode de novo. Eu testei: a semelhança entre gato e cachorro cai de 0,88 para 0,87, e a de gato e carro sobe de 0,40 para 0,47. Frases novas puxam os vetores um pouco. Agora invente frases só sobre frutas e veja se uma fruta passa a se parecer mais com outra do que com um gato.

CAPÍTULO 9: PREVER A PRÓXIMA PALAVRA



Chegamos à ideia mais simples e mais poderosa de toda a história dos assistentes de conversa: prever a próxima palavra. Um assistente que escreve parágrafos, responde dúvidas ou compõe poemas faz uma coisa só, repetidamente: olha o que já está escrito e escolhe a palavra que vem a seguir. Depois, a palavra escolhida entra no texto e ele repete. Palavra por palavra.

Você já usa isso todo dia

Abra o teclado do celular e digite “Vou chegar”. Em cima das teclas aparecem sugestões: “já”, “mais tarde”, “atrasado”. O teclado está prevendo a próxima palavra, com base no que milhões de pessoas costumam escrever depois de “vou chegar”. Os assistentes de IA fazem a mesma coisa, só que com uma memória muito maior e uma noção muito mais fina do contexto.

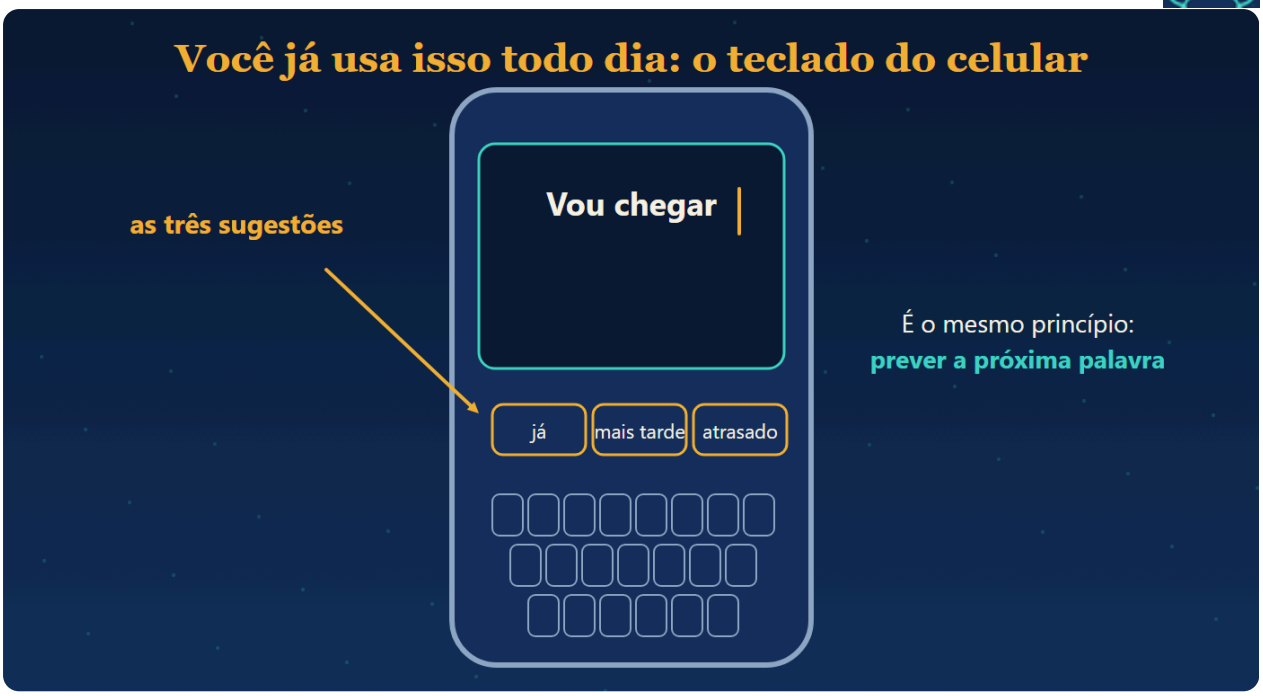


Figura 9.1 · Você já usa isso todo dia: o teclado do celular.



Uma história de pares de palavras

A ideia tem mais de um século. Em 1913, o matemático russo Andrei Markov contou, no poema *Eugênio Oneguin*, de Púchkin, quantas vezes uma vogal vinha depois de uma consoante, e a recíproca. Queria mostrar que a probabilidade de uma letra depende da anterior. Em 1948, Claude Shannon, o pai da teoria da informação, usou contagens parecidas para gerar textos que imitavam o inglês: com pares de letras e depois de palavras, sentenças saíam quase plausíveis, quase sem sentido. Cada método descende dessa ideia: o que vem depois depende do que veio antes.

Um gerador de brinquedo

Vamos construir o nosso. O programa lê um texto pequeno, de nove frases, e para cada palavra anota quais palavras vieram logo depois dela. Para gerar, começa por uma palavra, sorteia uma das que costumam vir depois dela, anota, e repete até chegar a um ponto final.

O sorteio respeita as repetições: se, no texto, depois de “o” veio “gato” quatro vezes e “jornal” uma vez, “gato” tem quatro vezes mais chance.

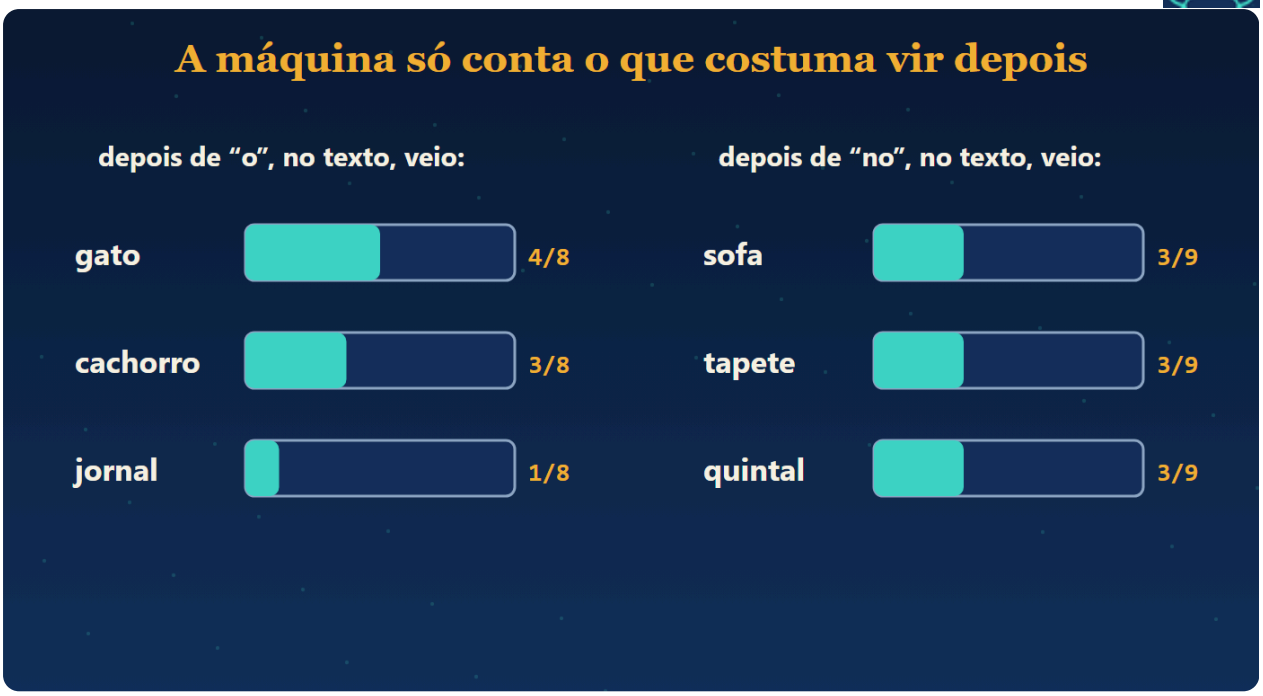


Figura 9.2 · O gerador só conta o que costuma vir depois.

Rode você mesmo



c09_proxima_palavra.py



```
# Prever a próxima palavra: um gerador de texto de brinquedo.
import random
from collections import defaultdict

texto = """o gato dorme no sofa . o gato bebe leite no sofa . o cachorro dorme no
tapete .
o cachorro come osso no quintal . o gato come peixe no quintal . o cachorro bebe agua
no tapete .
a menina lê um livro no sofa . a menina bebe leite no quintal . o gato lê o jornal no
tapete ."""

palavras = texto.split()
depois = defaultdict(list)            # para cada palavra, quem veio depois dela
for atual, seguinte in zip(palavras, palavras[1:]):
    depois[atual].append(seguinte)

print("Depois de 'o' já vi:", sorted(set(depois["o"])))
print("Depois de 'no' já vi:", sorted(set(depois["no"])))

random.seed(3)
for tentativa in range(3):
    palavra = "o"
    frase = [palavra]
    while palavra != "." and len(frase) < 12:
        palavra = random.choice(depois[palavra])
        frase.append(palavra)
    print(" ".join(frase))
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c09_proxima_palavra

o computador imprime

```
Depois de 'o' já vi: ['cachorro', 'gato', 'jornal']
Depois de 'no' já vi: ['quintal', 'sofa', 'tapete']
o cachorro bebe leite no tapete .
o jornal no sofa .
o gato bebe leite no tapete .
```

As primeiras duas linhas mostram o que o gerador aprendeu: depois de “o”, já viu “cachorro”, “gato” e “jornal”; depois de “no”, “quintal”, “sofá” e “tapete”. As três últimas são as frases que ele criou. A primeira, “o cachorro bebe leite no tapete”, tem sentido, e nenhuma frase do texto era assim. Ele combinou pedaços. A segunda, “o jornal no sofá”, é só um fragmento. A terceira, “o gato bebe leite no tapete”, está muito boa.



Figura 9.3 · Uma frase nasce palavra por palavra.

O gerador é simples e, mesmo assim, faz uma coisa que parece mágica: escreve frases novas, que não estavam no texto. Ele não entende nada de gatos. Só combina o que já viu, respeitando as vizinhanças.

Olhar uma palavra só é pouco

O nosso gerador tem um defeito grave: olha apenas a palavra anterior. Depois de “no”, ele sorteia sofá, tapete ou quintal, sem saber se o assunto era um livro, uma comida ou o gato. Veja: “a menina lê um livro no ____”. O ideal seria sofá. Mas o gerador só enxerga “no”, e pode dizer quintal.

Para melhorar, é preciso olhar mais palavras para trás. Dá para fazer isso aumentando a janela (olhar as duas, as três últimas), mas logo o método esbarra num muro: o número de combinações explode, e a maioria delas nunca aparece no texto de treino. Quem deu a saída foi a combinação de tudo o que vimos até aqui: vetores de palavras, redes neurais e a ideia de atenção, que o próximo capítulo explica.

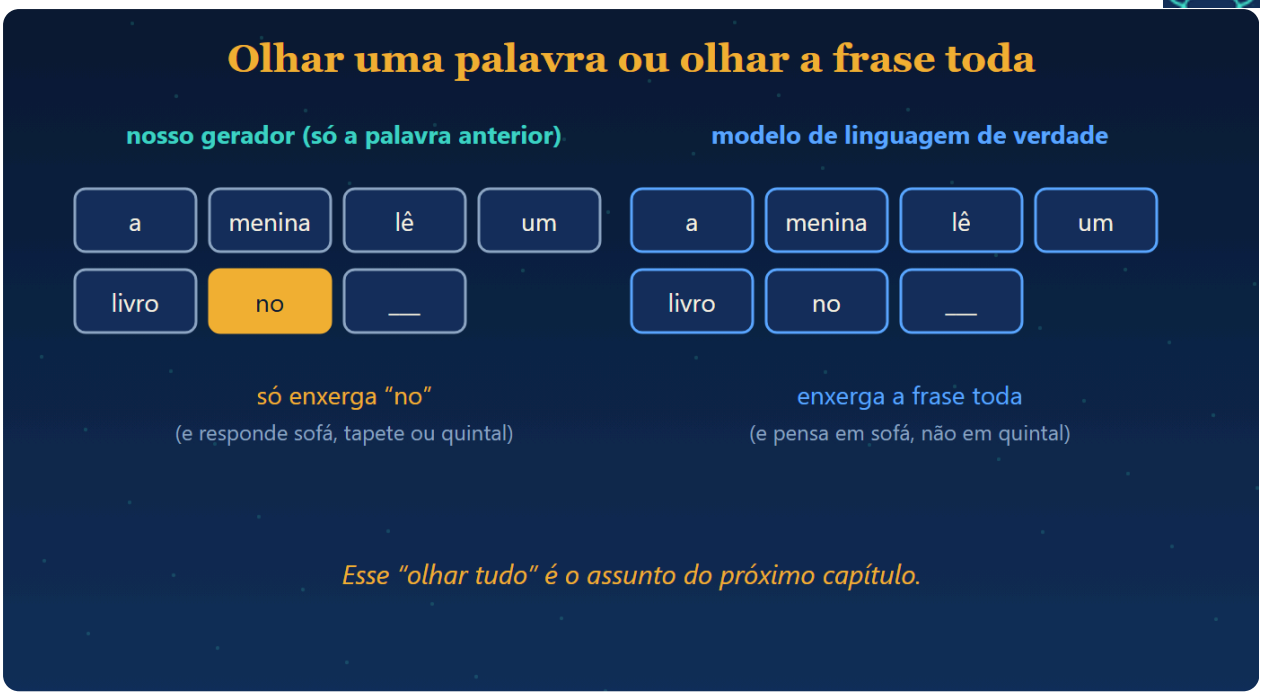


Figura 9.4 · Olhar uma palavra ou olhar a frase toda.

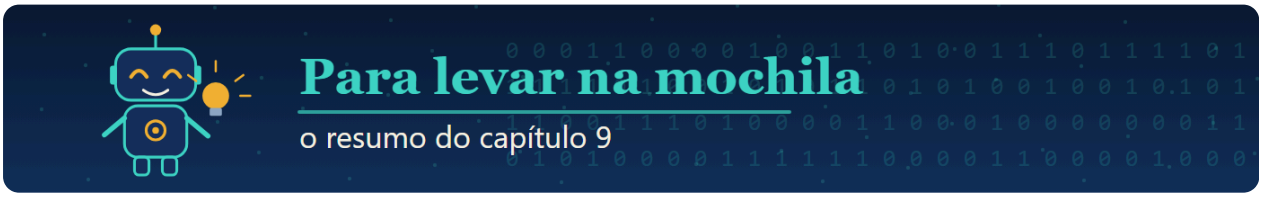


Por que prever a próxima palavra basta

Parece pouco, mas pense no que é preciso para prever bem. Para completar "a capital da França é ____", é preciso saber geografia. Para completar "ela chorou porque o filho tinha ____", é preciso um pouco de psicologia. Para completar a próxima linha de um código, é preciso entender a lógica do programa. Quem prevê bem a próxima palavra, em milhões de contextos, acaba aprendendo muita coisa sobre o mundo que as palavras descrevem. É a aposta que está por trás dos modelos de linguagem atuais.

E tem um efeito colateral que já vimos aparecer no nosso exemplo: o gerador pode escrever frases falsas com a mesma calma das verdadeiras. "O gato lê o jornal" saiu do nosso texto, e nenhum gato lê jornal. A máquina só prevê o que é provável; não confere se é verdade.

O que o Bit aprendeu

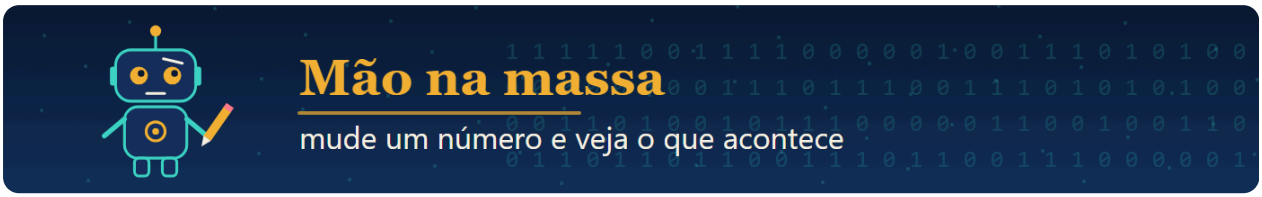


- Um assistente escreve prevendo a próxima palavra, uma de cada vez.



- O nosso gerador conta o que costuma vir depois e sorteia.
- Olhar só a palavra anterior é pouco; é preciso olhar o contexto.
- O que é provável não é necessariamente verdadeiro.

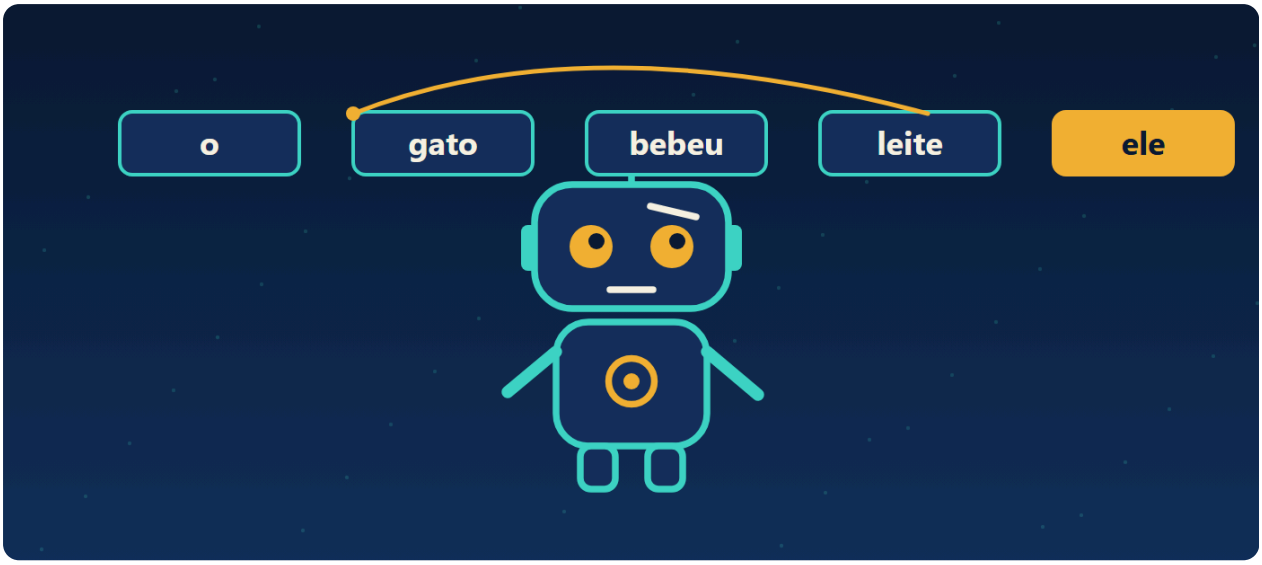
Experimente



Troque `random.seed(3)` por outro número e veja frases diferentes saírem do mesmo gerador. Depois acrescente ao texto duas frases suas, com o mesmo estilo, e veja se elas aparecem nas frases geradas. Se quiser um desafio: mude o código para que o gerador comece por “a” em vez de “o”.



CAPÍTULO 10: A ATENÇÃO



Leia esta frase: “O gato bebeu o leite porque ele tinha sede.” Quem tinha sede? O gato, claro. Você resolveu isso sem esforço, olhando de “ele” para trás e ligando a palavra ao “gato”. Agora troque o fim: “O gato bebeu o leite porque ele estava quente.” Aqui “ele” é o leite. A mesma palavra, outro dono. Um modelo de linguagem precisa fazer esse tipo de ligação o tempo todo, e foi para isso que inventaram a atenção.

O que é atenção

No gerador do capítulo anterior, cada palavra olhava só a anterior. A ideia da atenção é dar a cada palavra o poder de olhar todas as outras, e decidir quanto prestar atenção em cada uma. Quando o modelo processa “ele”, a palavra faz uma pergunta às outras: “quem aqui é um ser vivo que pode ter sede?”. As palavras respondem. Aquelas cuja resposta combina mais com a pergunta recebem mais atenção.



Figura 10.1 · Para entender “ele”, a palavra olha para todas as outras.

A ideia surgiu em 2014, em trabalhos de tradução automática (Dzmitry Bahdanau e colegas). E em 2017, um grupo de pesquisadores do Google, liderado por Ashish Vaswani, publicou um artigo de título ousado: *Attention Is All You Need*, “Atenção é tudo de que você precisa”. Apresentava uma arquitetura nova, o Transformer, que usava só atenção, sem os mecanismos mais pesados que existiam. Quase todos os modelos de linguagem famosos de hoje descendem dele; o “T” do GPT, aliás, é de *Transformer*.

Pergunta, etiqueta e conteúdo

A atenção funciona como uma biblioteca com três coisas. Cada palavra produz uma pergunta (em inglês *query*, Q), uma etiqueta que diz o que ela é (*key*, K) e um conteúdo que oferece (*value*, V). Para decidir onde olhar, a palavra compara a sua pergunta com a etiqueta de cada uma das outras. As que combinam mais ganham uma nota alta. Depois, o modelo mistura os conteúdos, na proporção das notas.



Figura 10.2 · Pergunta, etiqueta e conteúdo.

Três passos

É mais simples do que parece. Primeiro, cada palavra recebe uma nota: quanto a etiqueta dela combina com a pergunta. Segundo, as notas viram percentuais que somam 100%, com uma conta chamada softmax. Terceiro, mistura-se o conteúdo de todas, na proporção desses percentuais.

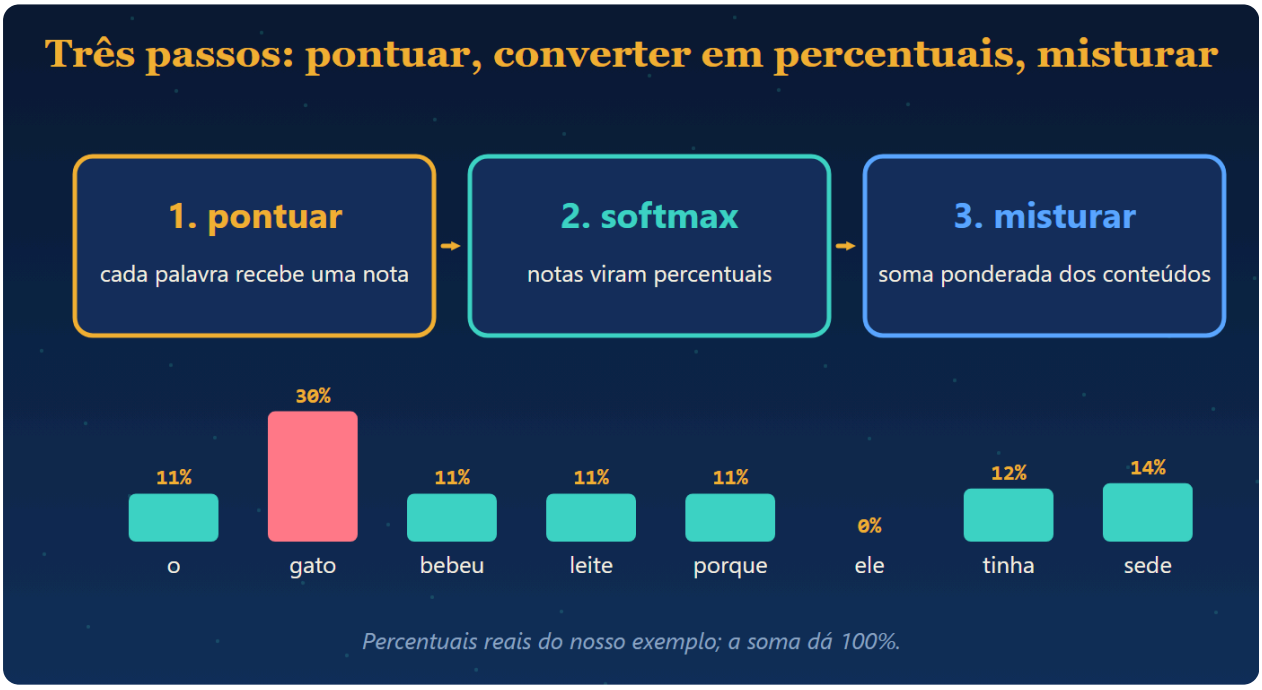


Figura 10.3 · Três passos: pontuar, transformar em percentuais, misturar.

Rode você mesmo

Vamos fazer os dois primeiros passos com a frase do gato. Para a conta ficar clara, escolhi à mão os números de cada palavra, em quatro dimensões: “é um ser vivo”, “é uma ação”, “é um líquido” e “aponta para alguém”. Num modelo de verdade, esses números não são escolhidos por ninguém: são aprendidos no treino, como os pesos da rede. Aqui, servem para você ver a conta.



c10_atencao.py

```
# Atenção: cada palavra olha para as outras e decide quanto prestar atenção em cada
uma.
import numpy as np

palavras = ["o", "gato", "bebeu", "leite", "porque", "ele", "tinha", "sede"]
# vetores escolhidos à mão para o exemplo (no modelo de verdade, eles são aprendidos)
# dimensões: [é um ser vivo, é uma ação, é um líquido, aponta para alguém]
v = np.array([
    [0.0, 0.0, 0.0, 0.0],   # o
    [1.0, 0.0, 0.0, 0.0],   # gato
    [0.0, 1.0, 0.0, 0.0],   # bebeu
    [0.0, 0.0, 1.0, 0.0],   # leite
    [0.0, 0.2, 0.0, 0.0],   # porque
    [0.9, 0.0, 0.0, 1.0],   # ele (aponta para um ser vivo)
    [0.1, 0.5, 0.0, 0.0],   # tinha
    [0.2, 0.0, 0.8, 0.0],   # sede
])

consulta = np.array([1.0, 0.0, 0.0, 1.0])     # "ele" pergunta: quem é um ser vivo?
notas = v @ consulta                           # o quanto cada palavra responde
notas[5] = -np.inf                             # "ele" não vale como resposta para si
mesmo
pesos = np.exp(notas) / np.exp(notas).sum()    # transforma em percentuais (softmax)

for p, w in zip(palavras, pesos):
    print(f"{p:7s} {w:5.1%} " + "#" * int(w * 60))
print("A palavra 'ele' presta mais atenção em:", palavras[int(np.argmax(pesos))])
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c10_atencao

o computador imprime



```
o       11.1% ######
gato    30.1% ##################
bebeu   11.1% ######
leite   11.1% ######
porque  11.1% ######
ele      0.0%
tinha   12.2% #######
sede    13.5% ########
A palavra 'ele' presta mais atenção em: gato
```

A palavra “ele” pergunta: “quem aqui é um ser vivo?”. O “gato” responde com a nota mais alta e recebe 30% da atenção; as outras dividem o resto, de 11% a 14% cada. A própria palavra “ele” ficou com 0, porque eu a tirei de propósito da competição: uma palavra que “aponta” para alguém não pode ser a resposta para si mesma. Num modelo real, não se faz esse remendo, e a conta é feita em muitas dimensões e muitas vezes, de modo paralelo, as chamadas “cabeças” de atenção.

Aviso de honestidade: o 30% é um número deste exemplo. Em um modelo de verdade, quase nunca se consegue apontar uma ligação tão limpa, nem dizer exatamente o que cada cabeça de atenção está fazendo. Os pesquisadores estudam isso, e muita coisa continua em aberto.

Empilhar blocos

Um Transformer faz isso muitas vezes, em camadas. Cada bloco tem uma etapa de atenção, onde as palavras trocam informação, e uma etapa de rede, como a do capítulo 6, onde cada palavra processa o que recebeu. O bloco se repete dezenas de vezes. Nas primeiras camadas, as palavras captam coisas simples, como a vizinhança e a gramática. Nas últimas, captam o sentido e a intenção. No fim, o resultado de cada posição é transformado numa lista de probabilidades das palavras que podem vir a seguir, e uma é sorteada, como no capítulo anterior.



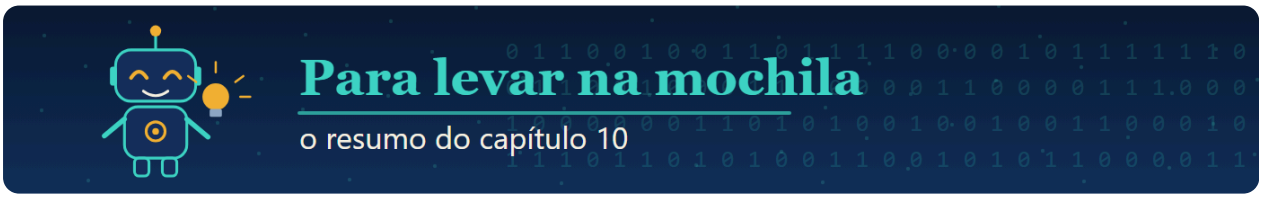
Figura 10.4 · O mesmo bloco, empilhado muitas vezes.



Por que isso mudou tudo

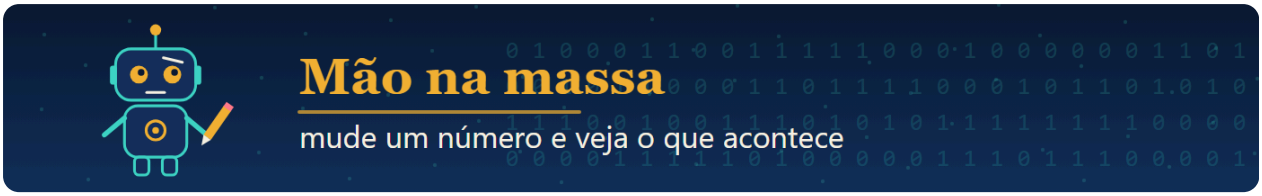
Duas razões. A primeira: a atenção deixa o modelo olhar para qualquer parte do texto, perto ou longe, com a mesma facilidade. A segunda é prática: ao contrário de modelos mais antigos, que liam uma palavra de cada vez, o Transformer pode processar todas as palavras de uma vez, em paralelo. Isso casa bem com as placas gráficas, os chips que os computadores usam para jogos, e permitiu treinar modelos com quantidades de texto nunca vistas antes.

O que o Bit aprendeu



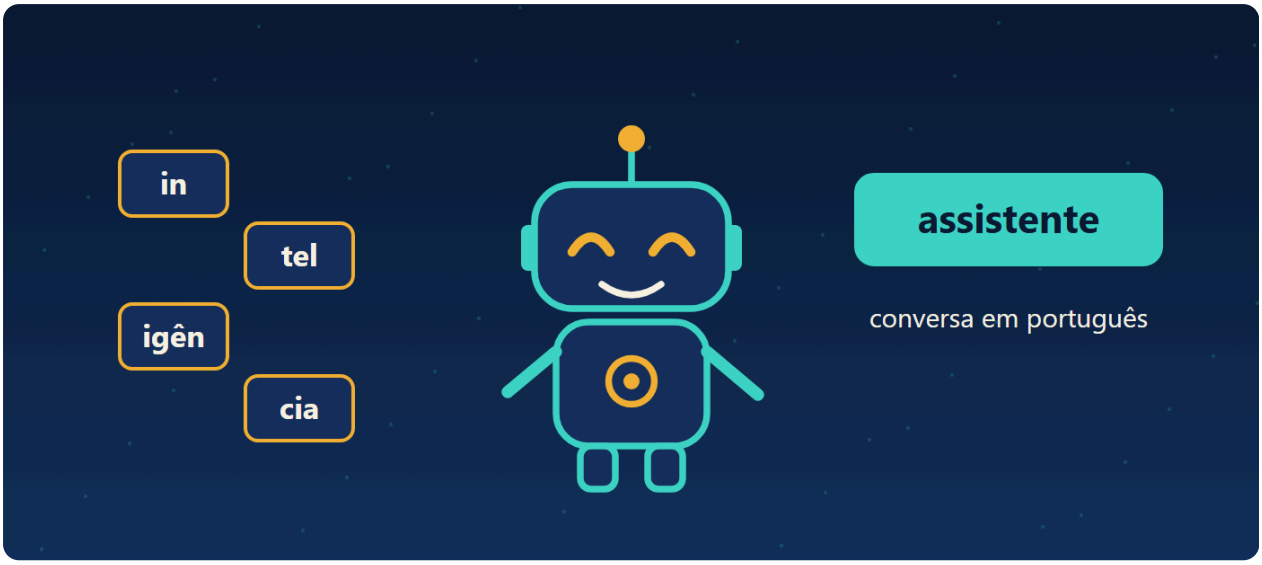
- Atenção é deixar cada palavra olhar todas as outras e decidir em quais prestar atenção.
- Cada palavra faz uma pergunta; as outras respondem com etiquetas; o conteúdo se mistura na proporção da resposta.
- O Transformer, de 2017, usa blocos de atenção e rede empilhados.
- Mesmo assim, o modelo só prevê a próxima palavra, uma de cada vez.

Experimente



Em `c10_atencao.py`, mude a consulta para `np.array([0.0, 1.0, 0.0, 0.0])`: agora a palavra pergunta "quem é uma ação?". Quem recebe mais atenção? Depois tente `[0.0, 0.0, 1.0, 0.0]`, "quem é um líquido?", e veja se a resposta faz sentido.

CAPÍTULO 11: DO MODELO AO ASSISTENTE



Um modelo que prevê a próxima palavra, por si só, não é um assistente. Se você escrever “Qual é a capital da França?”, um modelo bruto pode continuar com “Qual é a capital da Alemanha? Qual é a capital da Espanha?”, porque é isso que costuma vir depois de uma pergunta numa lista de perguntas. Para virar um assistente, que responde, é preciso mais do que prever. Este capítulo mostra as etapas que transformam um modelo de linguagem num assistente de conversa.

Etapa zero: o texto vira pedaços

Antes de tudo, o texto é cortado em pedaços chamados *tokens*. Um token pode ser uma palavra inteira, um pedaço de palavra ou até uma letra. Cada pedaço tem um número numa tabela, e é com esses números que o modelo trabalha. Uma palavra comum e curta costuma ser um token só; uma palavra rara ou comprida é cortada em vários.

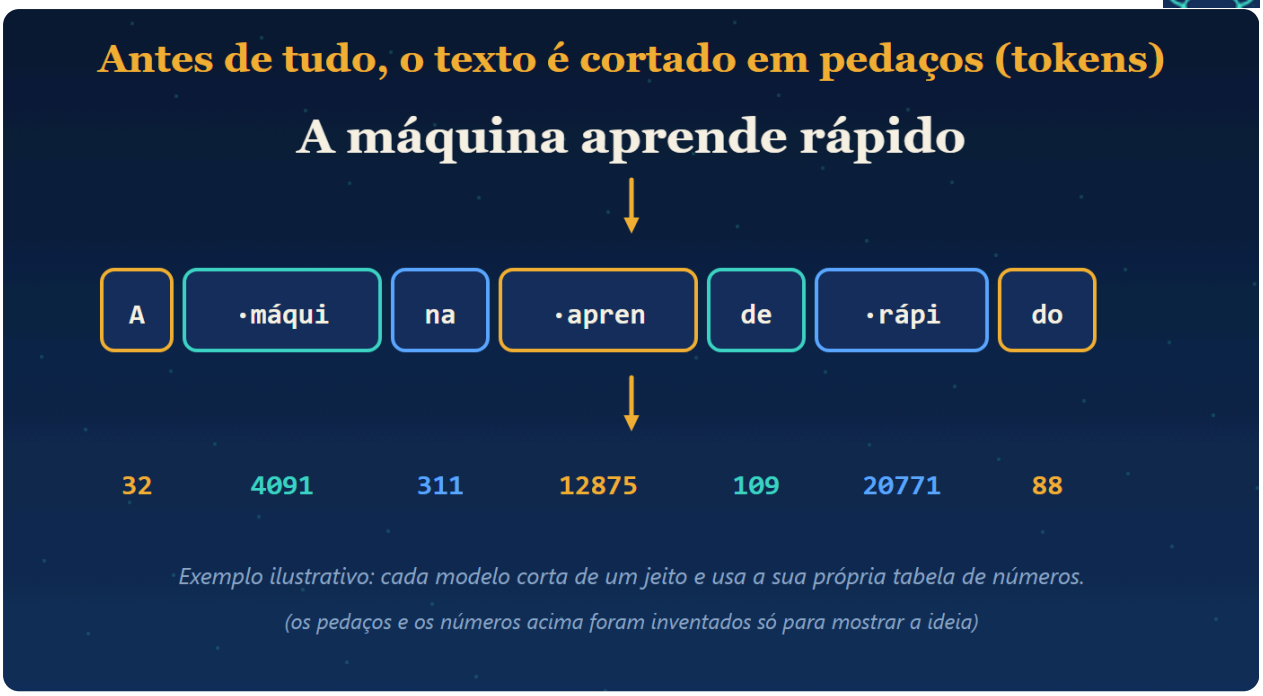


Figura 11.1 · Do texto aos números (ilustração, com números inventados).

O desenho é uma ilustração: cada modelo corta de um jeito e tem a sua própria tabela, e os números do desenho foram inventados só para mostrar a ideia. Quando se diz que um assistente lê tantos mil "tokens" por vez, é desses pedaços que se fala.

Etapa 1: o pré-treino

O modelo é treinado para prever a próxima palavra em uma quantidade imensa de texto: livros, artigos, páginas da internet, código de programas. É a mesma tarefa do capítulo 9, com a arquitetura do capítulo 10, em escala enorme. Aqui o modelo aprende o idioma, muita coisa sobre o mundo e também os defeitos e preconceitos dos textos. É a etapa mais cara. Os primeiros modelos da série GPT, da empresa OpenAI, apareceram em 2018 e foram crescendo; em 2020 o GPT-3 chamou a atenção do mundo por escrever textos fluentes.

Etapa 2: o ajuste com exemplos

O modelo pré-treinado sabe muito, mas não sabe conversar. Na segunda etapa, mostra-se a ele exemplos de conversa feitos por pessoas: uma pergunta e uma boa resposta, uma instrução e o resultado esperado, milhares de vezes. Ele continua prevendo a próxima

palavra, mas agora a partir de exemplos em que o que vem depois da pergunta é uma resposta. É como o aluno que, depois de ler bibliotecas inteiras, aprende com um professor o que é responder a uma pergunta.

Etapa 3: o ajuste com avaliação humana

Na terceira etapa, o modelo gera várias respostas à mesma pergunta e pessoas dizem qual preferem. A máquina é então ajustada para tender às respostas preferidas. A técnica se chama aprendizado por reforço com avaliação humana (em inglês, RLHF) e foi usada, em 2022, no InstructGPT e no ChatGPT. Foi um passo grande para o jeito educado, útil e cuidadoso dos assistentes atuais. O ChatGPT foi lançado em novembro de 2022 e, em poucos meses, passou a ser usado por milhões de pessoas.

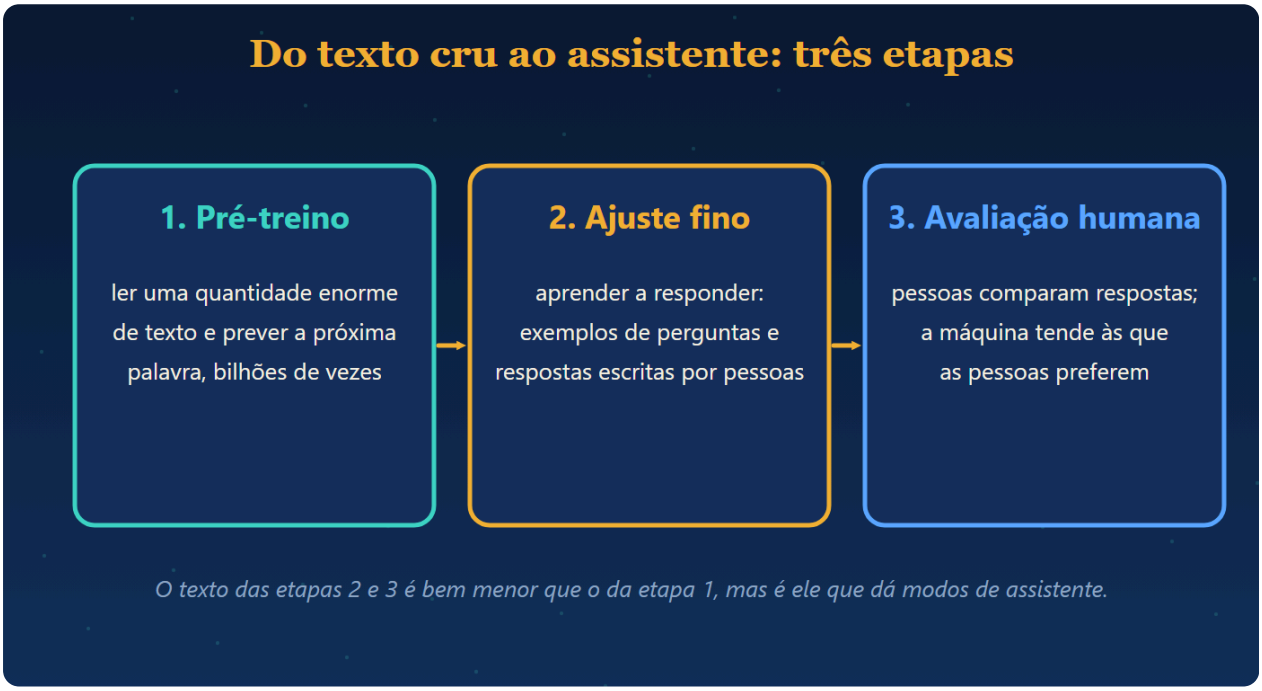


Figura 11.2 · Do texto cru ao assistente: três etapas.

Há variações entre empresas e modelos, e os detalhes de cada um nem sempre são públicos. Mas a ideia geral é essa: aprender o idioma, aprender a conversar, aprender o que as pessoas preferem.





Quando o assistente responde: o sorteio com temperatura

Já vimos que, a cada passo, o modelo calcula uma lista de probabilidades para a próxima palavra e sorteia. Existe um botão que controla esse sorteio, chamado temperatura. Temperatura baixa faz o modelo escolher quase sempre a palavra mais provável: respostas previsíveis e repetitivas. Temperatura alta dá chances às menos prováveis: respostas mais variadas e criativas, mas com mais risco de dizer bobagem.

Rode você mesmo

O programa a seguir usa quatro palavras candidatas, “sol”, “chuva”, “vento” e “neve”, com notas que escolhi à mão (3, 2, 1 e 0). Ele calcula as chances com três temperaturas e faz dez sorteios em cada.



c11_temperatura.py

```
# Temperatura: o assistente sorteia a próxima palavra, e a temperatura muda o sorteio.
import numpy as np

candidatas = ["sol", "chuva", "vento", "neve"]
pontos = np.array([3.0, 2.0, 1.0, 0.0])   # o que o modelo "acha" de cada uma

def probabilidades(temperatura):
    p = np.exp(pontos / temperatura)
    return p / p.sum()

rng = np.random.default_rng(5)
for t in (0.2, 1.0, 3.0):
    p = probabilidades(t)
    sorteio = rng.choice(candidatas, size=10, p=p)
    print(f"temperatura {t}: " + ", ".join(f"{c} {x:.0%}" for c, x in zip(candidatas,
p)))
    print("   dez sorteios:", " ".join(sorteio))
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c11_temperatura

o computador imprime



```
temperatura 0.2: sol 99%, chuva 1%, vento 0%, neve 0%
   dez sorteios: sol sol sol sol sol sol sol sol sol chuva
temperatura 1.0: sol 64%, chuva 24%, vento 9%, neve 3%
   dez sorteios: chuva sol sol neve vento chuva sol sol chuva sol
temperatura 3.0: sol 38%, chuva 28%, vento 20%, neve 14%
   dez sorteios: chuva sol neve sol vento neve sol neve neve sol
```

Com temperatura 0,2, “sol” tem 99% de chance, e saiu em nove dos dez sorteios. Com 1,0, o “sol” cai para 64% e as outras aparecem. Com 3,0, as chances ficam quase parelhas (38%, 28%, 20% e 14%), e o sorteio vira uma loteria. É o mesmo modelo, com a mesma lista de notas, mudando só o botão.

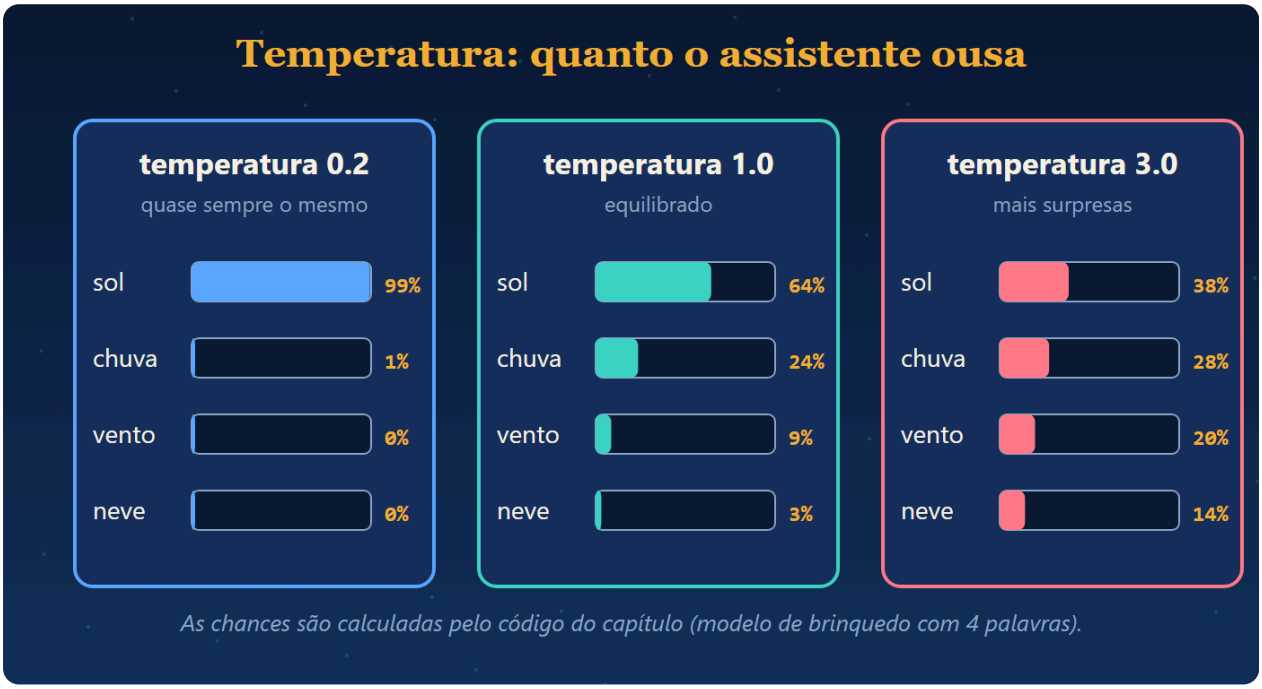


Figura 11.3 · Temperatura: quanto o assistente ousa.

A janela de contexto: quanto o modelo lê de uma vez

Um assistente não guarda memória como uma pessoa. A cada resposta, ele lê de novo tudo o que a conversa já tem, até um limite de tamanho. Esse limite se chama janela de contexto. Dentro da janela, o modelo “lembra” de tudo. Fora, não existe. Se uma conversa é longa demais, as primeiras mensagens saem da janela e o assistente passa a se comportar como se nunca tivessem sido ditas. Cada modelo tem o seu limite.

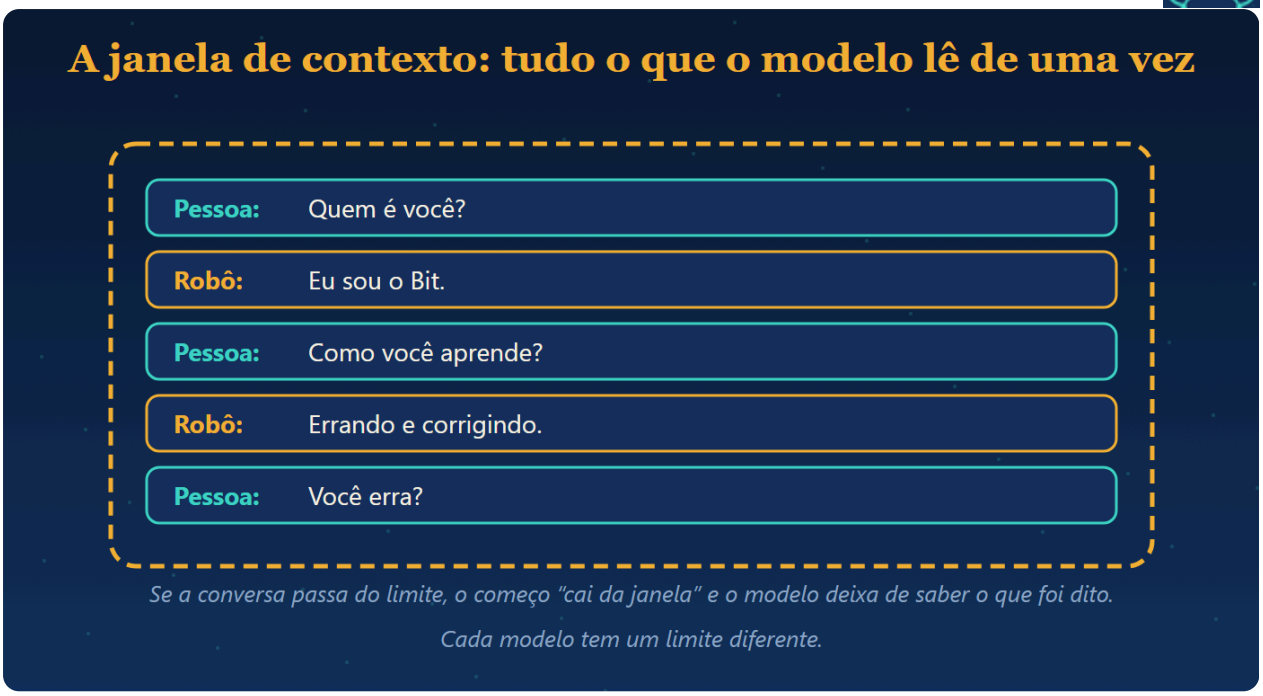
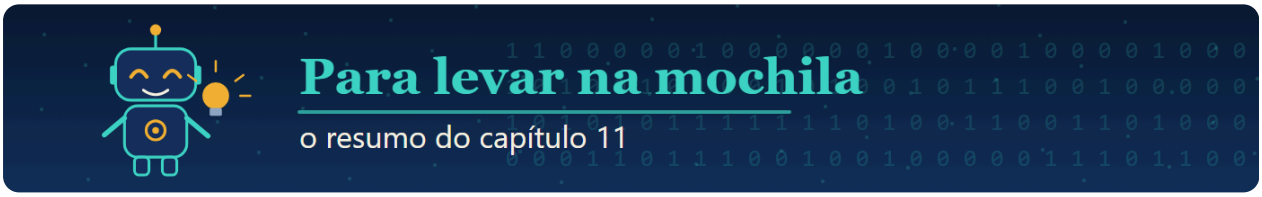


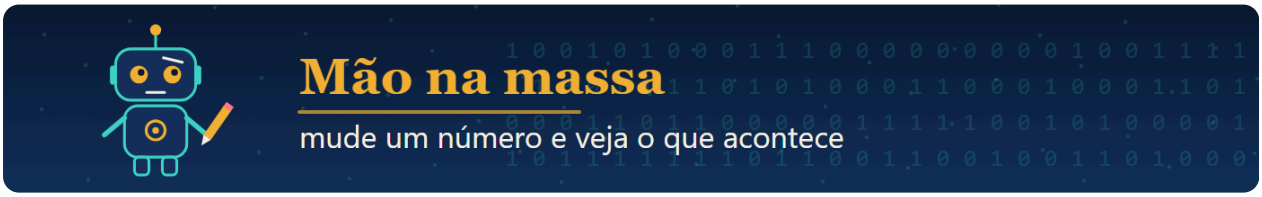
Figura 11.4 · A janela de contexto: tudo o que o modelo lê de uma vez.

O que o Bit aprendeu



- O texto é cortado em pedaços (tokens) e vira números.
- Um assistente passa por etapas: pré-treino em muito texto, ajuste com exemplos de conversa e ajuste pela preferência de pessoas.
- A temperatura controla o quanto o sorteio da próxima palavra arrisca.
- A janela de contexto é tudo o que o modelo lê a cada resposta.

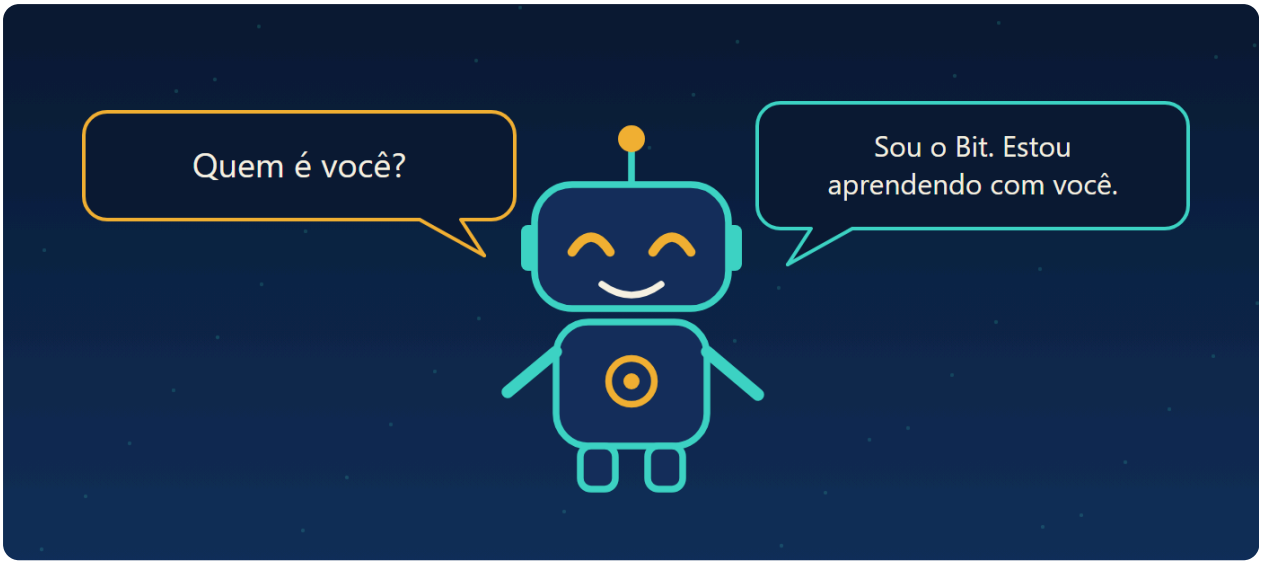
Experimente



Troque as temperaturas `(0.2, 1.0, 3.0)` por `(0.1, 0.5, 10)`. O que acontece com as chances? Depois mude os pontos `[3.0, 2.0, 1.0, 0.0]` para `[1.0, 1.0, 1.0, 1.0]`: com notas iguais, a temperatura importa?



CAPÍTULO 12: UM CHAT DE VERDADE



Juntamos todas as peças. A máquina aprende errando e corrigindo (capítulo 4). Faz isso com neurônios em camadas (5 e 6). Trabalha com números, inclusive os que representam palavras (3 e 8). Prevê a próxima palavra (9), olhando o contexto com atenção (10). E vira assistente depois de várias etapas de treino (11). Falta só uma coisa: a conversa. Neste último capítulo, vamos montar um chat que guarda o que foi dito e responde em sequência, e depois falaremos do que um bom usuário precisa saber.

O truque da memória

Como é que o assistente lembra do que você falou há três mensagens? A resposta é quase decepcionante: ele não lembra. A cada pergunta, o programa que cuida do chat pega a conversa inteira, de cima a baixo, e a envia de novo ao modelo, junto com a nova mensagem. O modelo lê tudo como se fosse um texto só e escreve a continuação. A “memória” é uma lista que cresce, guardada pelo programa, e não pelo modelo.

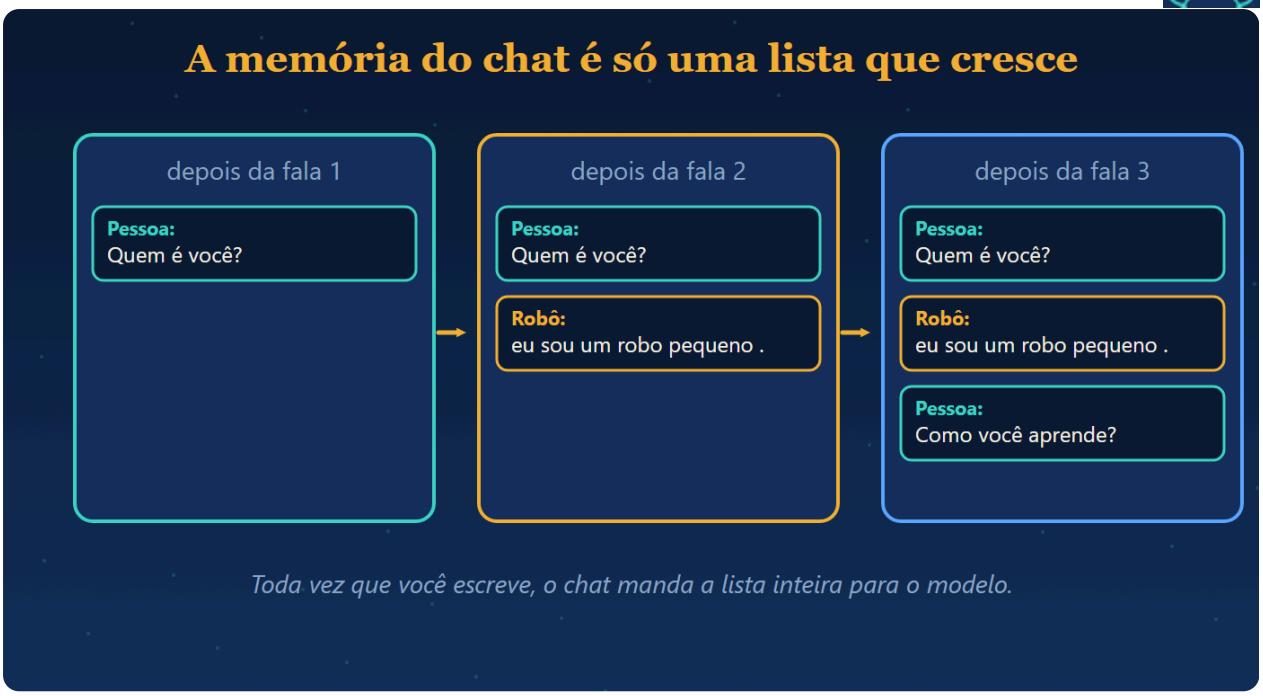


Figura 12.1 · A memória do chat é só uma lista que cresce.

Esse texto que vai ao modelo a cada vez se chama prompt. É a pergunta, mais o histórico, mais as instruções de comportamento que o programa decidiu incluir.



Figura 12.2 · O laço de um chat.



Rode você mesmo: um chat de brinquedo

O programa abaixo é um chat completo, mas de brinquedo: o "modelo" é o gerador de palavras do capítulo 9, treinado num texto de poucas frases sobre o próprio robô. Ele só olha a última fala da pessoa (um modelo de verdade lê o histórico todo), e troca "você" por "eu" para responder. O que importa aqui não é a qualidade da resposta, é a estrutura: a lista `historico`, o `prompt` montado a cada vez e a função `conversar`.



c12_chat.py



```
# Um chat completo, de brinquedo: guarda a conversa e monta o "prompt" a cada
resposta.
import random
from collections import defaultdict

# --- o "modelo": o gerador de palavras do capítulo 9, treinado num texto pequeno ---
texto = """eu sou um robo pequeno . eu aprendo errando e corrigindo . eu gosto de
aprender com voce .
eu erro muito . eu erro e depois eu corrijo . voce pode me perguntar o que quiser .
eu posso errar e voce pode me corrigir . aprender e errar e tentar de novo ."""
palavras = texto.split()
depois = defaultdict(list)
for a, b in zip(palavras, palavras[1:]):
    depois[a].append(b)

TROCAS = {"voce": "eu", "você": "eu", "erra": "erro", "aprende": "aprendo"}   # "você"
vira "eu"

def modelo(prompt):
    # este modelo de brinquedo só lê a última fala da pessoa (um modelo de verdade lê
tudo)
    fala = prompt.split("Pessoa:")[-1].split("\n")[0].lower().replace("?", "")
    conhecidas = [TROCAS.get(p, p) for p in fala.split()]
    conhecidas = [p for p in conhecidas if p in depois and p != "."]
    palavra = conhecidas[0] if conhecidas else "eu"
    resposta = [palavra]
    while palavra != "." and len(resposta) < 14:
        palavra = random.choice(depois[palavra])
        resposta.append(palavra)
    return " ".join(resposta)

# --- o chat: a memória é só uma lista que cresce ---
historico = []
def conversar(fala):
    historico.append("Pessoa: " + fala)
    prompt = "\n".join(historico) + "\nRobô:"
    resposta = modelo(prompt)
    historico.append("Robô: " + resposta)
    return resposta

random.seed(2)
for fala in ["Quem é você?", "Como você aprende?", "Você erra?"]:
    print("Pessoa:", fala)
    print("Robô:  ", conversar(fala))
print("--- tudo o que o chat guardou (o prompt da próxima vez) ---")
print("\n".join(historico) + "\nRobô:")
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c12_chat



o computador imprime

```
Pessoa: Quem é você?
Robô:   eu posso errar e corrigindo .
Pessoa: Como você aprende?
Robô:   eu posso errar e voce pode me perguntar o que quiser .
Pessoa: Você erra?
Robô:   eu erro e tentar de aprender com voce pode me corrigir .
--- tudo o que o chat guardou (o prompt da próxima vez) ---
Pessoa: Quem é você?
Robô: eu posso errar e corrigindo .
Pessoa: Como você aprende?
Robô: eu posso errar e voce pode me perguntar o que quiser .
Pessoa: Você erra?
Robô: eu erro e tentar de aprender com voce pode me corrigir .
Robô:
```

As respostas estão longe de ser boas. “eu posso errar e corrigindo.” quase faz sentido; “eu erro e tentar de aprender com voce pode me corrigir.” já embaralhou tudo. Era de esperar: o gerador só olha a palavra anterior. Mas a estrutura está inteira: a pessoa fala, o chat guarda, monta o prompt, chama o modelo, guarda a resposta. No fim, o programa mostra tudo o que guardou, a lista que seria enviada na próxima vez.

Trocando o brinquedo por um modelo de verdade

Para trocar o gerador por um modelo de verdade, o programa muda pouco. Em vez de calcular a resposta em casa, ele a pede, pela internet, a uma empresa que oferece o modelo. O exemplo abaixo usa a biblioteca da Anthropic, a criadora do Claude (as outras empresas têm bibliotecas parecidas). Ele **não foi executado aqui**, porque exige uma conta e uma chave de acesso, que são pessoais e pagas por uso. Está no livro para você ver como é pouca a diferença.



c12b_api.py



```
# O mesmo chat com um modelo de verdade (exemplo: não roda sem a sua chave de acesso).
# Instale com:  pip install anthropic
# A chave fica na variável de ambiente ANTHROPIC_API_KEY. Nunca escreva a chave dentro
do arquivo.
import anthropic

cliente = anthropic.Anthropic()          # lê a chave do ambiente
historico = []                           # a memória da conversa: uma lista

while True:
    fala = input("Você: ")
    if fala.strip() == "":
        break
    historico.append({"role": "user", "content": fala})
    resposta = cliente.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=500,
        system="Você é um assistente simpático que explica tudo em português
simples.",
        messages=historico,
    )
    texto = resposta.content[0].text
    historico.append({"role": "assistant", "content": texto})
    print("Assistente:", texto)
```

Rode este exemplo sem instalar nada: multiversodigital.com.br/Biografia/python-lab/?ex=c12b_api

Observe os pontos que já conhece: o `historico` é uma lista que cresce; a cada fala, o programa a envia inteira ao modelo; a resposta volta e entra na lista. A linha `system` é a instrução de comportamento: o jeito de o assistente falar.

E uma regra de segurança, que vale em qualquer serviço desse tipo: a chave de acesso é como a senha do seu banco. Nunca a escreva dentro do arquivo do programa, nunca a publique, nunca a envie para ninguém. Guarde-a numa variável de ambiente, como o exemplo faz, e não a deixe em lugares públicos. Quem tem a sua chave gasta o seu dinheiro.



O que a máquina não faz

O assistente parece saber de tudo, e é aí que moram os riscos. Vale conhecer os principais, que vêm de tudo o que vimos.

Ele inventa, com toda a confiança

O modelo prevê palavras prováveis, não verdades. Se você perguntar por algo que ele não sabe, ele pode inventar um fato, uma data, um nome, uma citação, um livro, uma lei, com a mesma segurança com que diz uma verdade. Em inglês se diz *hallucination*, alucinação. Não é mentira, pois não há intenção. É só uma consequência de a máquina escolher o que soa certo. No desenho a seguir, o Bit foi perguntado por algo que não existe, o seu próprio nascimento, e respondeu direitinho.

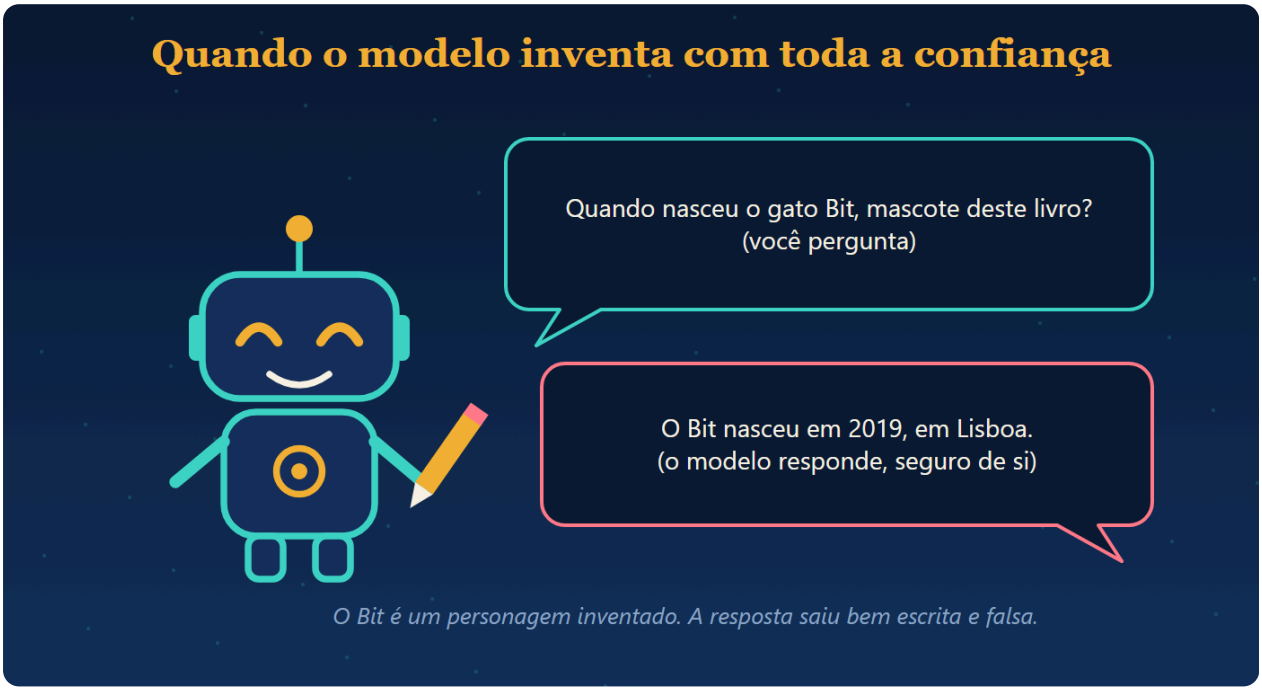


Figura 12.3 · Quando o modelo inventa com toda a confiança.

Ele herda os defeitos do que leu

Como vimos no capítulo 8, os modelos aprendem com textos de gente, e herdam o que há neles: erros, estereótipos, o ponto de vista de quem mais escreveu. Isso pode aparecer em respostas sobre pessoas, profissões, países, culturas. Quem usa o assistente precisa ler com olhar crítico, como lê qualquer fonte.

Sua conversa não é um diário secreto

O que você digita vai para os servidores de uma empresa. Cada serviço tem regras próprias sobre guardar e usar as conversas, e você deve ler as do serviço que usa. Como regra de prudência, nunca escreva senhas, números de documentos, dados de saúde ou segredos de outras pessoas, a não ser que saiba exatamente para onde vão.

Ele não decide por você

Quando a resposta importa, para a saúde, o dinheiro, a lei ou a vida de alguém, o assistente pode ajudar a pensar, resumir, dar ideias, mas quem confere e decide é você, e quem tem de responder pela decisão é você também.

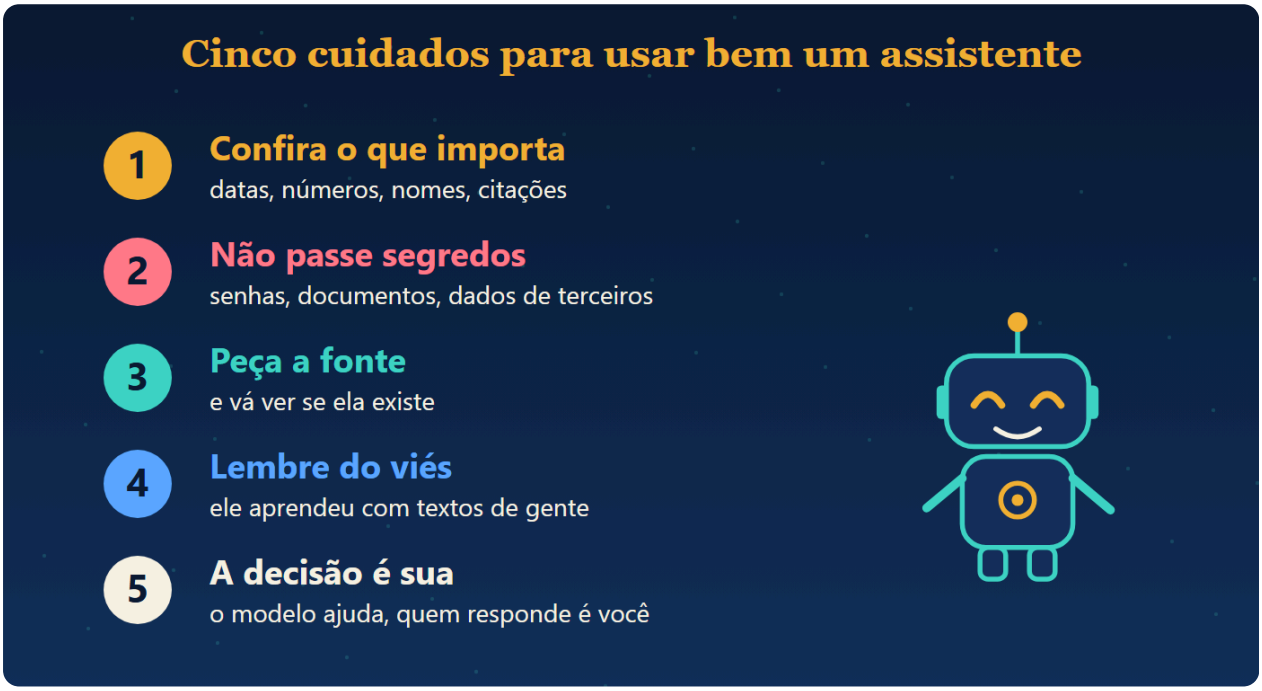


Figura 12.4 · Cinco cuidados para usar bem um assistente.



Um modo de usar que funciona

O melhor jeito de usar um assistente é tratá-lo como um colega muito lido, rápido e às vezes distraído: ótimo para rascunhar, explicar, comparar, organizar ideias e sugerir caminhos; ruim como única fonte de um fato. Peça que ele explique o raciocínio. Peça fontes, e confira se existem. Faça a mesma pergunta de duas maneiras e veja se a resposta muda. E, quando algo parecer bom demais, desconfie.

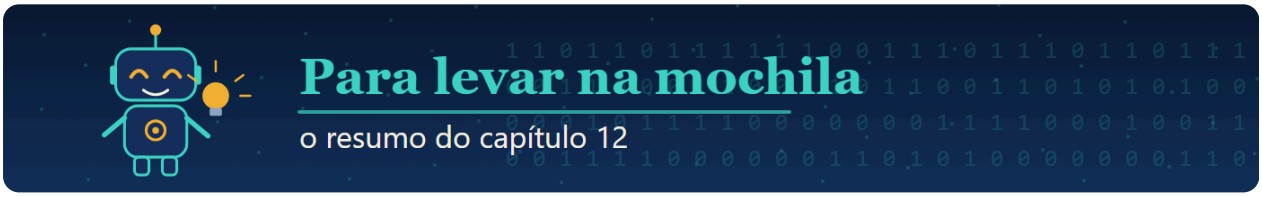


O fim do começo

Você percorreu um longo caminho. Começou com um programa que decide spam por regras e chegou a um chat que guarda a conversa. Viu uma máquina errar e corrigir uma reta, ganhar um neurônio, uma rede, olhos para ver dígitos, números para as palavras, atenção e, por fim, a fala. Em nenhum momento houve mágica. Há números, contas, exemplos, erro e correção, muita repetição, e uma quantidade enorme de texto escrito por gente.

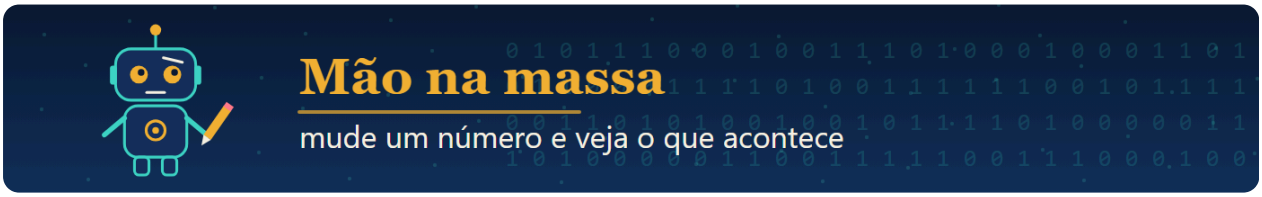
É por isso que vale entender. Uma tecnologia que ninguém entende vira superstição: ou se teme demais, ou se confia demais. Quem entende como ela funciona sabe para que serve, sabe quando duvidar e sabe fazer perguntas melhores. O Bit continua aprendendo. Você também.

O que o Bit aprendeu



- A memória de um chat é uma lista que o programa guarda e reenvia a cada vez.
- O modelo de verdade entra no lugar do gerador de brinquedo; a estrutura do chat é a mesma.
- O modelo prevê o que soa provável e pode inventar fatos com confiança.
- Confira o que importa, proteja seus dados e lembre que a decisão é sua.

Experimente

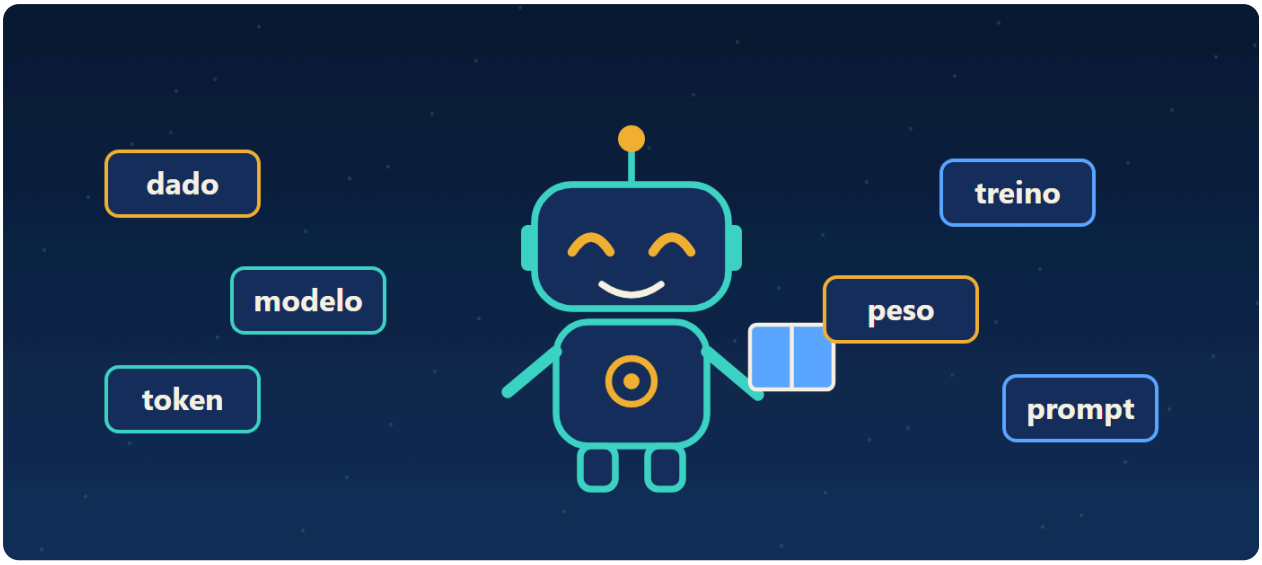


No chat de brinquedo, acrescente ao `texto` três frases suas sobre o robô, por exemplo “eu gosto de ler” ou “eu moro no computador”. Faça novas perguntas e veja se as respostas passam a falar disso. Depois, com um assistente de verdade, peça duas vezes a mesma



explicação de um assunto que você conhece bem, e confira as duas respostas com o seu conhecimento: onde elas acertam e onde escorregam?

GLOSSÁRIO



Palavras que aparecem no livro, explicadas em linguagem de gente.

Atenção

Mecanismo que deixa cada palavra olhar todas as outras de um texto e decidir em quais prestar mais atenção. Base do Transformer (capítulo 10).

Aprendizado de máquina

Fazer a máquina descobrir as regras a partir de exemplos, em vez de alguém escrevê-las à mão (capítulo 1).

Camada escondida

Camada de neurônios entre a entrada e a saída de uma rede. É ela que permite aprender coisas que não são retas (capítulo 6).



Descida do gradiente

Método de aprender: a cada passo, olhar para que lado o erro diminui e andar um pouco nessa direção (capítulo 4).



Embedding (vetor de palavra)

Lista de números que resume o significado de uma palavra pelas companhias que ela tem (capítulo 8).



Erro

A diferença entre o que a máquina previu e o que era certo. Aprender é reduzir o erro (capítulo 4).



Janela de contexto

Tudo o que o modelo lê de uma vez ao responder. O que está fora dela, para ele, não existe (capítulo 11).



Modelo

O resultado do treino: a máquina com os seus pesos já ajustados. É o que se usa depois, para responder.



Neurônio artificial

Peça que multiplica as entradas por pesos, soma e compara com um limite (capítulo 5).



Peso

Número que diz o quanto uma entrada importa. É a memória da máquina: tudo o que ela aprendeu está nos pesos (capítulo 5).



Pré-treino

A primeira etapa de um assistente: ler muito texto e prever a próxima palavra (capítulo 11).



Prompt

O texto inteiro que vai ao modelo a cada resposta: a sua pergunta, o histórico e as instruções (capítulo 12).



Retropropagação

Jeito de levar o erro da saída de volta pela rede, para que cada peso se corrija na proporção da sua culpa (capítulo 6).



Sobreajuste

Quando a máquina decora os exemplos de treino em vez de aprender; vai bem no treino e mal no teste (capítulo 7).



Temperatura

Botão que controla o quanto o assistente arrisca na escolha da próxima palavra (capítulo 11).



Teste

Exemplos que a máquina nunca viu, usados para medir se ela aprendeu de verdade (capítulo 7).



Token

Pedaço de texto (palavra, parte de palavra ou letra) com um número na tabela do modelo (capítulo 11).



Transformer

Arquitetura de rede neural de 2017 que usa blocos de atenção e rede empilhados. O “T” do GPT (capítulo 10).



Treino

O processo em que a máquina palpita, mede o erro e corrige os pesos, muitas vezes (capítulo 4).



Viés (duas coisas diferentes)

No neurônio, o número fixo que se soma à conta (capítulo 5). Na sociedade, a tendência de um modelo a repetir preconceitos que existem nos textos em que aprendeu (capítulos 8 e 12).



COMO INSTALAR E RODAR TUDO



Este apêndice leva você do zero até o primeiro exemplo funcionando. Se algo der errado, volte aqui.



Se preferir não instalar nada

Todos os exemplos do livro rodam também no site, dentro do navegador, no **laboratório de Python**: multiversodigital.com.br/Biografia/python-lab/. Escolha o exemplo na lista, aperte "Rodar" e veja o resultado. Dá para mudar o código e rodar de novo, e para baixar o arquivo `.py`. Na versão do livro na tela, cada exemplo tem um quadro "Rodar este exemplo aqui" logo abaixo do código. Na primeira vez, o laboratório baixa o Python e as bibliotecas (uns 10 MB), e a página precisa de internet. O seu código roda dentro do seu próprio navegador e não é enviado a lugar nenhum. As instruções a seguir servem para quem quiser o Python no computador.



1. Instalar o Python

Windows. Entre em python.org, baixe o instalador do Python 3 e execute. Na primeira tela, antes de clicar em "Install Now", marque a caixa *Add python.exe to PATH*. É a causa de metade dos problemas de quem começa. **Mac.** O mesmo site tem o instalador para Mac.



Linux. Quase todas as distribuições já trazem o Python 3; se não, instale pelo gerenciador de pacotes.

Para conferir, abra o Prompt de Comando (no Windows, digite “cmd” no menu Iniciar) ou o Terminal e escreva `python --version`. Deve aparecer algo como “Python 3.13.3”. No Mac e no Linux, o comando pode ser `python3`.



2. Instalar as bibliotecas

Só alguns capítulos precisam de bibliotecas extras. Instale ambas de uma vez:

```
pip install numpy scikit-learn
```

A `numpy` faz contas com tabelas de números (capítulos 6, 8, 10 e 11). A `scikit-learn` traz as imagens de dígitos e a rede do capítulo 7. Os capítulos 1 a 5, 9 e 12 (o chat de brinquedo) usam só o Python puro. A instalação baixa alguns megabytes da internet e leva um ou dois minutos.



3. Criar e rodar um arquivo

Crie uma pasta, por exemplo `ia-com-python`. Abra o Bloco de Notas (ou outro editor de texto simples), copie o código de um exemplo do livro e salve na pasta com o nome do exemplo, como `c04_reta.py`. Cuide para que o arquivo termine mesmo em `.py`, e não em `.py.txt`. Depois, no Prompt de Comando, entre na pasta e rode:

```
cd ia-com-python
python -X utf8 c04_reta.py
```

O `-X utf8` faz o Python escrever os acentos de forma correta no Windows. Se aparecerem letras estranhas no lugar deles, é isso que está faltando. Salve também o arquivo em formato UTF-8 (no Bloco de Notas, ao salvar, escolha “UTF-8” em “Codificação”).

O resultado do exemplo aparece na tela, igual ao do livro. Os exemplos com sorteio (capítulos 9, 11 e 12) usam uma semente fixa, então saem iguais para quem usa a mesma versão do Python e da numpy; em outras versões, o sorteio pode dar frases diferentes. Isso é normal.



4. Problemas comuns

O que aparece	O que costuma ser
“python não é reconhecido como comando”	faltou marcar “Add python.exe to PATH” na instalação; reinstale marcando
“No module named numpy” (ou sklearn)	faltou o `pip install numpy scikit-learn`
“IndentationError”	o recuo está errado: use quatro espaços, sempre os mesmos
“SyntaxError”	falta dois-pontos, parêntese ou aspas fechando; veja a linha indicada
“can't open file”	você não está na pasta do arquivo; use `cd` para entrar nela
ConvergenceWarning (capítulo 7)	só um aviso: a rede ainda ia melhorar um pouco; pode ignorar



5. Os arquivos dos exemplos

Todos os exemplos do livro têm o nome `cNN_assunto.py`, em que NN é o número do capítulo: `c01_spam_regras.py`, `c02_python.py`, `c03_numeros.py`, `c04_reta.py`, `c05_neuronio.py`, `c06_rede_xor.py`, `c07_digitos.py`, `c07b_decorar.py`, `c08_palavras.py`, `c09_proxima_palavra.py`, `c10_atencao.py`, `c11_temperatura.py`, `c12_chat.py` e `c12b_api.py`. Este último só roda com uma chave de acesso, como explica o capítulo 12.



6. Quando quiser ir além

Depois deste livro, o caminho natural é aprofundar uma ponta de cada vez: aprender mais Python (há cursos gratuitos de qualidade), fazer pequenos projetos com dados seus (uma planilha de gastos, uma lista de filmes) e experimentar assistentes de IA com os cuidados do capítulo 12. O mais importante você já tem: sabe, em linhas gerais, como a coisa funciona. O resto é prática.