Fluxo do Bigrama

Geração de Nomes com Modelos de Bigramas e PyTorch

Este guia integra os conceitos de **Verossimilhança Negativa**, **normalização de probabilidades** e **otimização de modelos de bigramas**, explicando como eles se conectam para gerar nomes que imitam padrões de linguagem.

---

1. A Teoria por Trás do Modelo

Um **modelo de bigrama** prevê o próximo caractere com base apenas no anterior, usando a **probabilidade condicional**.

Normalização

A probabilidade de um caractere j vir depois de um caractere i (P_i, j) é calculada dividindo a contagem de sua co-ocorrência (C_i, j) pela soma de todas as contagens a partir de i.

$$P_{i,j} = \frac{C_{i,j}}{\sum_k C_{i,k}}$$

Em PyTorch, essa normalização é feita de forma vetorizada em uma matriz de contagens `N`, onde cada linha é normalizada para somar 1.

```
P = N.float()
P = P / P.sum(1, keepdim=True) # keepdim=True é essencial para broadcasting
```

Função de Perda (Loss)

Para avaliar a qualidade do modelo, usamos a **Negative Log-Likelihood (NLL)**. Um modelo é considerado bom quando atribui altas probabilidades aos bigramas corretos no dataset, resultando em um NLL baixo. A NLL média é a função de perda que o modelo tenta minimizar durante o treinamento.

$$\text{NLL}_{\text{avg}} = \frac{-\sum \log(p_i)}{N}$$

Para evitar a perda infinita ($-log(0)$), o **model smoothing** adiciona contagens fictícias à matriz.

---

2. Do Texto à Rede Neural

Para treinar um modelo de bigramas usando redes neurais, o texto é transformado em dados numéricos.

1. **Dataset de Bigramas**: As palavras são divididas em pares de caracteres (`(X, Y)`). Um token especial (`.`) marca o início e o fim.
2. **Codificação One-Hot**: Os caracteres de entrada (`X`) são convertidos em vetores **one-hot**, que são compatíveis com redes neurais.
3. **Logits**: O vetor one-hot é multiplicado pela matriz de pesos da rede (`W`), produzindo os **logits**, que são pontuações brutas.
4. **Softmax**: Os logits são transformados em uma **distribuição de probabilidade** (valores entre 0 e 1, somando 1) através da função softmax.
5. **Perda**: A perda do modelo é calculada a partir da probabilidade que o modelo atribui ao caractere correto (`Y`), usando NLL.
6. **Otimização**: Os pesos `W` são ajustados via **otimização por gradiente** para minimizar a perda.

---

3. Geração de Nomes na Prática

Após o treinamento, o modelo pode gerar nomes que parecem reais usando amostragem probabilística.

1. **Ponto de Partida**: O processo começa com o token `.` (início da palavra).
2. **Previsão**: A linha de probabilidades correspondente ao token atual é selecionada.
3. **Amostragem**: `torch.multinomial` sorteia o próximo caractere com base na distribuição de probabilidades.
4. **Loop**: O novo caractere é adicionado ao nome e o ciclo se repete.
5. **Fim da Geração**: O processo para quando o token `.` (fim da palavra) é sorteado novamente.

O resultado é um nome com estrutura fonética plausível, mas que não existe no dataset.

---

4. Exemplo Simplificado

Mini-Dataset: `ana`, `bob`, `bea`
Vocabulário: `.`=0, `a`=1, `b`=2, `e`=3, `n`=4, `o`=5

- **Pares (X, Y):** `(0,1), (1,4), (4,1), (1,0)`, etc.
- **Treinamento**: Uma pequena rede neural (matriz `W` 6x6) é treinada para minimizar a NLL usando os pares.
- **Geração**:
  - Começa com `.` (índice 0).
  - Rede gera probabilidades para o próximo caractere (por exemplo, `P(a)=0.5, P(b)=0.4, ...`).
  - `torch.multinomial` sorteia `a` (índice 1). Nome gerado: `a`.
  - Próxima entrada: `a` (índice 1). Rede gera probabilidades para o próximo caractere (por exemplo, `P(n)=0.8`).
  - `torch.multinomial` sorteia `n` (índice 4). Nome gerado: `an`.

- O processo continua até o token `.` ser sorteado, completando o nome, por exemplo, `ana`.

1 Como a rede aprende nomes

- A rede aprende **padrões entre letras** (quais letras normalmente vêm depois de outras).
- Reconhece inícios (`.` → `a`), sequências (`th` → `e` / `a`) e finais (`ia` , `son`).
- Pode ser feita com uma **matriz simples** (bigram) ou uma **rede pequena**.
- Para gerar nomes, usamos **softmax + amostragem**, escolhendo letras com base em probabilidades.

2 Mini-exemplo

- Dataset: `ana` , `anna` , `anael` .
- Padrões frequentes: início → `a` ; `a` → `n` ; `n` → `a` ou `e` .
- Com tokens de início/fim `.` , a rede aprende probabilidades.
- Pode gerar nomes novos como: `anal` , `ane` .

3 Passos do código de Karpathy

1. Lê nomes e adiciona token `.` no início e fim.
2. Cria pares `(anterior, próximo)` e conta frequências em uma matriz `W` .
3. Para gerar nomes: começa com `.` e escolhe o próximo caractere com base nas probabilidades da matriz.
4. A rede aprende padrões de **transição entre letras**, não o significado dos nomes.
5. Mesmo com rede pequena, captura dependências básicas do dataset.

4 Como prever a próxima letra

1. Converte letra anterior em **índice** no vocabulário.
2. Cria um vetor **one-hot** (1 para a letra atual, 0 para as demais).
3. Multiplica pelo peso `W` → obtém **logits** (valores brutos).

4. Aplica **softmax** → transforma logits em probabilidades.
5. Escolhe a próxima letra (pode ser a mais provável ou amostrada).
6. Repete até chegar no token final `.`.

5 Exemplo numérico rápido

Vocabulário: `{'.','a','b'}`, `prev='a'` → one-hot `[0,1,0]`.

Matriz `W`:

Prev\Next	'.'	'a'	'b'
'.'	0.5	1.0	-0.5
'a'	0.2	-0.1	0.7
'b'	-0.3	0.8	0.5

Multiplicação `[0,1,0] × W` → `[0.2, −0.1, 0.7]`.
Softmax → `[0.295, 0.218, 0.486]`.
Resultado: próxima letra = `'b'`.

6 Mini-dataset e pares

Nomes: `ana`, `bob`, `bea` (+ token início/fim `.`)

Pares extraídos:

```
.→a, a→n, n→a, a→.
.→b, b→o, o→b, b→.
.→b, b→e, e→a, a→.
```

7 Treinamento da rede

```
import torch
```

```
xs = torch.tensor([0,1,4,1, 0,2,5,2, 0,2,3,1])  # letra anterior
ys = torch.tensor([1,4,1,0, 2,5,2,0, 2,3,1,0])  # próxima letra
correta

vocab_size = 6
W = torch.randn((vocab_size, vocab_size), requires_grad=True)
optimizer = torch.optim.SGD([W], lr=50)

# Treinamento rápido
for step in range(5):
    logits = W[xs]
    probs = logits.exp() / logits.exp().sum(1, keepdims=True)
    loss = -probs[range(xs.nelement()), ys].log().mean()
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print(f"Passo {step} - Loss: {loss.item()}")
```

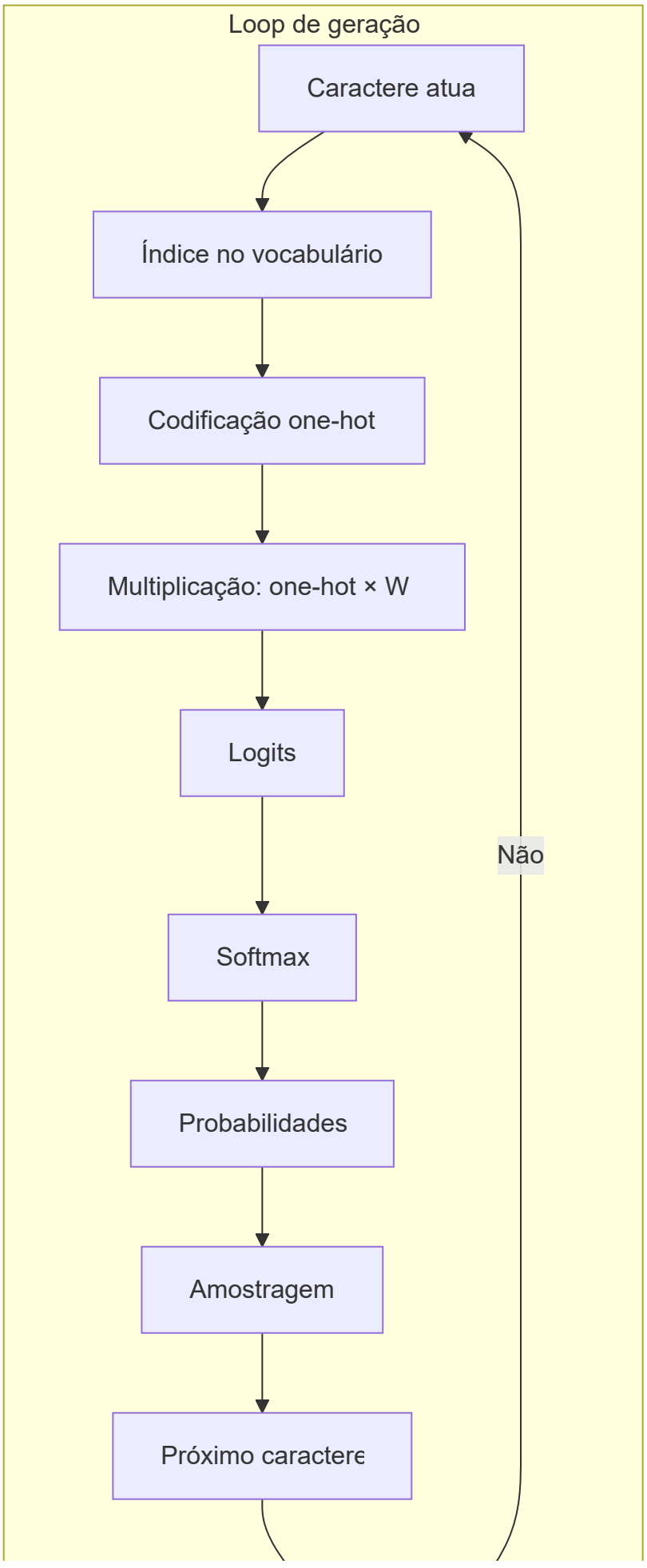
8 Gerando nomes

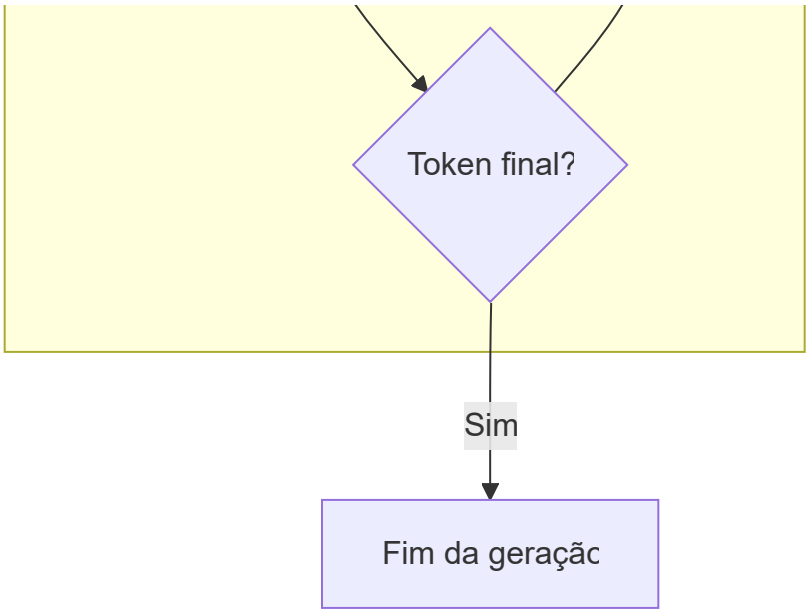
```
g = torch.Generator().manual_seed(42)
vocab = [".", "a", "b", "e", "n", "o"]

for _ in range(3):
    out = []
    ix = 0  # começa com token '.'
    while True:
        logits = W[ix]
        probs = logits.exp() / logits.exp().sum()
        ix = torch.multinomial(probs, 1, generator=g).item()
        if ix == 0: break
        out.append(vocab[ix])
    print("".join(out))
```

- Loop até encontrar o token final `.`
- Probabilidades vêm da linha correspondente de `W`
- Exemplos gerados: `bob`, `ana`, `bea`

Fluxo


Loop de geração
Caractere atua
Índice no vocabulário
Codificação one-hot
Multiplicação: one-hot × W
Logits
Softmax
Probabilidades
Amostragem
Próximo caractere
Não



Referencia

- https://colab.research.google.com/drive/1YIfmkftLrz6MPTOO9Vwqrop2Q5llHIGK?usp=sharing#scrollTo=TQUMmgRrdRIA
- https://www.youtube.com/watch?v=TCH_1BHY58I&t=2487s
- https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf