02-TCD-tokenizer

August 14, 2025

---

1 Tópicos em Ciência de Dados

1.0.1 Pontifícia Universidade Católica de Campinas

1.0.2 Prof. Dr. Denis Mayr Lima Martins

Conteúdo baseado no livro Build a Large Language Model From Scratch de Sebastian Rasckha

---

Supplementary code for the Build a Large Language Model From Scratch book by Sebastian Raschka Code repository: https://github.com/rasbt/LLMs-from-scratch

2 Tokenização

[15]:

```python
import os
import urllib.request

if not os.path.exists("the-verdict.txt"):
    url = ("https://raw.githubusercontent.com/rasbt/"
           "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
           "the-verdict.txt")
    file_path = "the-verdict.txt"
    urllib.request.urlretrieve(url, file_path)
```

[16]:

```python
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

print("Número de caracteres no texto:", len(raw_text))
```

```
Número de caracteres no texto: 20479
```

[17]:

```python
print(raw_text[:99])
```

```
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow
enough--so it was no
```

3 Separando tokens

[18]:
```
import re

text = "Hello, world. Is this-- a test?"

result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

[19]:
```
preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]
print(preprocessed[:30])
```

```
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a', 'cheap',
'genius', '--', 'though', 'a', 'good', 'fellow', 'enough', '--', 'so', 'it',
'was', 'no', 'great', 'surprise', 'to', 'me', 'to', 'hear', 'that', ',', 'in']
```

4 Número total de tokens

[20]:
```
print(len(preprocessed))
```

```
4690
```

5 Trabalhando com Token IDs

[21]:
```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)

print(vocab_size)
```

```
1130
```

[22]:
```
vocab = {token:integer for integer,token in enumerate(all_words)}
```

[23]:
```
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

```
('!', 0)
('"', 1)
("'", 2)
('(', 3)
(')', 4)
(',', 5)
('--', 6)
```

```
('.', 7)
(':', 8)
(';', 9)
('?', 10)
('A', 11)
('Ah', 12)
('Among', 13)
('And', 14)
('Are', 15)
('Arrt', 16)
('As', 17)
('At', 18)
('Be', 19)
('Begin', 20)
('Burlington', 21)
('But', 22)
('By', 23)
('Carlo', 24)
('Chicago', 25)
('Claude', 26)
('Come', 27)
('Croft', 28)
('Destroyed', 29)
('Devonshire', 30)
('Don', 31)
('Dubarry', 32)
('Emperors', 33)
('Florence', 34)
('For', 35)
('Gallery', 36)
('Gideon', 37)
('Gisburn', 38)
('Gisburns', 39)
('Grafton', 40)
('Greek', 41)
('Grindle', 42)
('Grindles', 43)
('HAD', 44)
('Had', 45)
('Hang', 46)
('Has', 47)
('He', 48)
('Her', 49)
('Hermia', 50)
```

6 Criando um Tokenizer simples

[24]:

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

    def encode(self, text):
        """Transforma texto em token IDs"""
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)

        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        """Transforma token IDs devolta em texto"""
        text = " ".join([self.int_to_str[i] for i in ids])
        # Replace spaces before the specified punctuations
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text)
        return text
```

[25]:

```
tokenizer = SimpleTokenizerV1(vocab)

text = """"It's the last he painted, you know,"
           Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)
```

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 1108,
754, 793, 7]
```

[26]:

```
tokenizer.decode(ids)
```

[26]:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardonable
pride.'
```

[27]:

```
tokenizer.decode(tokenizer.encode(text))
```

[27]:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardonable
pride.'
```

7 Adicionando tokens especiais

- Alguns tokenizadores utilizam tokens especiais para ajudar o modelo de linguagem (LLM) a ter contexto adicional.

- Entre esses tokens especiais estão:
    - `[BOS]` – *beginning of sequence* (início da sequência) marca o começo do texto;
    - `[EOS]` – *end of sequence* (fim da sequência) indica onde o texto termina (é usado, por exemplo, para concatenar vários textos não relacionados, como dois artigos diferentes da Wikipédia ou dois livros diferentes);
    - `[PAD]` – *padding*. Quando treinamos LLMs com tamanho de lote maior que 1, incluímos múltiplos textos de comprimentos distintos; o token de padding preenche os textos mais curtos para atingir o comprimento máximo, garantindo que todos tenham o mesmo tamanho;
    - `[UNK]` – representa palavras que não estão no vocabulário.

  Observação: O GPT-2 não precisa desses tokens (`[BOS]`, `[EOS]`, `[PAD]`, `[UNK]`) porque ele usa um tokenizador de *byte-pair encoding* (BPE) que fragmenta palavras em unidades sub-palavra, evitando a necessidade de um token `<UNK>`.

[28]:

```python
tokenizer = SimpleTokenizerV1(vocab)

text = "Hello, do you like tea. Is this-- a test?"

tokenizer.encode(text)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[28], line 5
      1 tokenizer = SimpleTokenizerV1(vocab)
      3 text = "Hello, do you like tea. Is this-- a test?"
----> 5 tokenizer.encode(text)

Cell In[24], line 13, in SimpleTokenizerV1.encode(self, text)
      8 preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
     10 preprocessed = [
     11     item.strip() for item in preprocessed if item.strip()
     12 ]
---> 13 ids = [self.str_to_int[s] for s in preprocessed]
     14 return ids

Cell In[24], line 13, in <listcomp>(.0)
      8 preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
     10 preprocessed = [
     11     item.strip() for item in preprocessed if item.strip()
     12 ]
---> 13 ids = [self.str_to_int[s] for s in preprocessed]
     14 return ids
```

```
KeyError: 'Hello'
```

- O trecho acima gera um erro porque a palavra “Hello” não está contida no vocabulário.

- Para tratar esses casos, podemos adicionar tokens especiais como `"<|unk|>"` ao vocabulário para representar palavras desconhecidas.

- Como já estamos estendendo o vocabulário, vamos adicionar outro token chamado `"<|endoftext|>"` para marcar o fim do texto

[29]:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])

vocab = {token:integer for integer,token in enumerate(all_tokens)}
```

[30]:

```
len(vocab.items())
```

[30]:

```
1132
```

[31]:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

8 Tokenizer com tokens especiais

[32]:

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.:;?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        preprocessed = [
            item if item in self.str_to_int
            else "<|unk|>" for item in preprocessed
        ]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        # Replace spaces before the specified punctuations
        text = re.sub(r'\s+([,.:;?!"()\'])', r'\1', text)
        return text
```

[33]:

```
tokenizer = SimpleTokenizerV2(vocab)

text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."

text = " <|endoftext|> ".join((text1, text2))

print(text)
```

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of the palace.
```

[34]:

```
tokenizer.encode(text)
```

[34]:

```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131, 7]
```

[35]:

```
tokenizer.decode(tokenizer.encode(text))
```

[35]:

```
'<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of the <|unk|>.'
```

9 Sliding window em texto

[36]:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

```
4690
```

[37]:

```
enc_sample = enc_text[50:]
```

[38]:

```
context_size = 4

x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]

print(f"x: {x}")
print(f"y:      {y}")
```

```
x: [568, 115, 1066, 727]
y:      [115, 1066, 727, 988]
```

10 Simulando a predição da próxima palavra

[39]:

```python
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]

    print(context, "---->", desired)
```

```
[568] ----> 115
[568, 115] ----> 1066
[568, 115, 1066] ----> 727
[568, 115, 1066, 727] ----> 988
```

[40]:

```python
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]

    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

```
in ----> a
in a ----> villa
in a villa ----> on
in a villa on ----> the
```

[41]:

```python
import torch
print("PyTorch version:", torch.__version__)
```

```
PyTorch version: 2.8.0
```

[42]:

```python
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        # Tokenize the entire text
        token_ids = tokenizer.encode(txt, allowed_special={"<|endoftext|>"})
        assert len(token_ids) > max_length, "Number of tokenized inputs must at␣
 ↪least be equal to max_length+1"

        # Use a sliding window to chunk the book into overlapping sequences of␣
 ↪max_length
        for i in range(0, len(token_ids) - max_length, stride):
            input_chunk = token_ids[i:i + max_length]
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))
    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]
```

[ ]:

```
import tiktoken

def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):

    # Initialize the tokenizer
    tokenizer = tiktoken.get_encoding("gpt2") # BPE: https://www.bpe-visualizer.
 ↪com/

    # Create dataset
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)

    # Create dataloader
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )

    return dataloader
```

[44]:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
```

[45]:

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False
)

data_iter = iter(dataloader)
first_batch = next(data_iter)
print(first_batch)
```

```
[tensor([[  40,  367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

[46]:

```
second_batch = next(data_iter)
print(second_batch)
```

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

[47]:

```
dataloader = create_dataloader_v1(raw_text, batch_size=8, max_length=4,␣
 ↪stride=4, shuffle=False)

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

```
Inputs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])

Targets:
 tensor([[  367,  2885,  1464,  1807],
        [ 3619,   402,   271, 10899],
        [ 2138,   257,  7026, 15632],
        [  438,  2016,   257,   922],
        [ 5891,  1576,   438,   568],
        [  340,   373,   645,  1049],
        [ 5975,   284,   502,   284],
        [ 3285,   326,    11,   287]])
```

11 Criando token embeddings

[48]:

```
# Assume 4 inputs com ids 2, 3, 5 e 1 (depois da tokenização)
input_ids = torch.tensor([2, 3, 5, 1])
```

[49]:

```
# Utilizando um vocabulário de 6 palavras, criamos embeddings de␣
 ↪dimensionalidade 3

vocab_size = 6
output_dim = 3

torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

[51]:

```
# Matriz de pesos 6x3
print(embedding_layer.weight)
```

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178,  1.5810,  1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-1.1589,  0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

[53]:

```
# Converte o token de id 3 para um vetor 3-d de embeddings
print(embedding_layer(torch.tensor([3])))
```

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

[55]:

```
# Embeddings para todas as entradas
print(embedding_layer(input_ids))
```

```
tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

12 Enconding de posições

[56]:

```
# Configurando o vocabulário para o mesmo tamanho do BPE
vocab_size = 50257
output_dim = 256

token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

[ ]:

```
# Batch size de 8 com 4 tokens cada, isso resulta em um tensor 8 x 4 x 256

max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
    stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
```

[63]:

```
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

```
Token IDs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [   284,  3285,   326,    11]])

Inputs shape:
 torch.Size([8, 4])
```

[64]:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
#print(token_embeddings)
```

```
torch.Size([8, 4, 256])
```

[65]:

```
# GPT-2 usa posições absolutas para os embeddings
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
# print(pos_embedding_layer.weight)
```

[66]:

```
pos_embeddings = pos_embedding_layer(torch.arange(max_length))
print(pos_embeddings.shape)
# print(pos_embeddings)
```

```
torch.Size([4, 256])
```

[68]:

```
# Embeddings de entrada usados em um LLM:
# Basta somar o embedding do token e o embedding posicional.
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
# print(input_embeddings)
```

```
torch.Size([8, 4, 256])
```