

Let's build the GPT Tokenizer

Based on Andrej Karpathy's [Let's build the GPT Tokenizer](#), a 02h13min34sec video published on 2024-02-21.

☰ Table of Contents

Let's build the GPT Tokenizer
Tokenization
Decoding
Encoding
Forced splits using regex patterns (GPT series)
Special tokens
minbpe exercise

```
1 begin
2     using PlutoUI
3     PlutoUI.TableOfContents(indent=true, depth=4, aside=true)
4 end
```

```
1 using Printf
```

```
1 using JSON
```

Tokenization

Tokenization is at the heart of much weirdness of LLMs. Do not brush it off.

- Why can't LLM spell words? **Tokenization**.
- Why can't LLM do super simple string processing tasks like reversing a string? **Tokenization**.
- Why is LLM worse at non-English languages (e.g. Japanese)? **Tokenization**.
- Why is LLM bad at simple arithmetic? **Tokenization**.
- Why did GPT-2 have more than necessary trouble coding in Python? **Tokenization**.
- Why did my LLM abruptly halt when it sees the string `<|endof text|>` **Tokenization**.
- What is this weird warning I get about a trailing whitespace? **Tokenization**.
- Why the LLM break if I ask it about "SolidGoldMagikarp"? **Tokenization**.
- Why should I prefer to use YAML over JSON with LLMs? **Tokenization**.
- Why is LLM not actually end-to-end language modeling? **Tokenization**.
- What is the real root of suffering? **Tokenization**.

Good tokenization web app: <https://tiktokenizer.vercel.app>

```
base_sentence = "こんにちは (Hello, Good day in Japanese)"
```

```
1 base_sentence = "こんにちは (Hello, Good day in Japanese)"
```

```
► [12371, 12395, 12385, 12399, 32, 40, 72, 101, 108, 108, 111, 44, 32, 71, 111, 111, 100, 32, 100, 97, 121, 32, 1
```

```
1 [Int(x) for x ∈ base_sentence]
```

```
► [0xe3, 0x81, 0x93, 0xe3, 0x81, 0xab, 0xe3, 0x81, 0xa1, 0xe3, 0x81, 0xaf, 0x20, 0x28, 0x48, 0x65, 0x6c, 0x6c, 0
```

```
1 (collect ∘ codeunits)(base_sentence)
```

```

1 begin
2   # text from https://www.reedbeta.com/blog/programmers-intro-to-unicode/
3   text = ""Unicode! 🐍🐍🐍🐍? UNICODE! 😊 The very name strikes fear and awe into the hearts of
4   programmers worldwide. We all know we ought to “support Unicode” in our software (whatever that
5   means—like using wchar_t for all the strings, right?). But Unicode can be abstruse, and diving into
6   the thousand-page Unicode Standard plus its dozens of supplementary annexes, reports, and notes can
7   be more than a little intimidating. I don’t blame programmers for still finding the whole thing
8   mysterious, even 30 years after Unicode’s inception.""
9
10  tokens = (collect ◦ codeunits)(text)           # raw bytes utf-8
11  tokens = map(x -> Int(x), tokens) |> collect   # convert to a list of integers in range 0..255 for
12  convenience
13
14  println("---\n$(text)\nlength: $(length(text))\n---\n")
15  println("$(tokens)\nlength: $(length(tokens))")
16 end

```

```

--
Unicode! 000000? UNICODE! 😊 The very name strikes fear and awe into the hearts of programmer
s worldwide. We all know we ought to “support Unicode” in our software (whatever that means—like u
sing wchar_t for all the strings, right?). But Unicode can be abstruse, and diving into the thousa
nd-page Unicode Standard plus its dozens of supplementary annexes, reports, and notes can be more
than a little intimidating. I don’t blame programmers for still finding the whole thing mysterious
, even 30 years after Unicode’s inception.
length: 533
--
[239, 188, 181, 239, 189, 142, 239, 189, 137, 239, 189, 131, 239, 189, 143, 239, 189, 132, 239, 18
9, 133, 33, 32, 240, 159, 133, 164, 240, 159, 133, 157, 240, 159, 133, 152, 240, 159, 133, 146, 24
0, 159, 133, 158, 240, 159, 133, 147, 240, 159, 133, 148, 226, 128, 189, 32, 240, 159, 135, 186, 2
26, 128, 140, 240, 159, 135, 179, 226, 128, 140, 240, 159, 135, 174, 226, 128, 140, 240, 159, 135,
168, 226, 128, 140, 240, 159, 135, 180, 226, 128, 140, 240, 159, 135, 169, 226, 128, 140, 240, 159
, 135, 170, 33, 32, 240, 159, 152, 132, 32, 84, 104, 101, 32, 118, 101, 114, 121, 32, 110, 97, 109
, 101, 32, 115, 116, 114, 105, 107, 101, 115, 32, 102, 101, 97, 114, 32, 97, 110, 100, 32, 97, 119
, 101, 32, 105, 110, 116, 111, 32, 116, 104, 101, 32, 104, 101, 97, 114, 116, 115, 32, 111, 102, 3
2, 112, 114, 111, 103, 114, 97, 109, 109, 101, 114, 115, 32, 119, 111, 114, 108, 100, 119, 105, 10
0, 101, 46, 32, 87, 101, 32, 97, 108, 108, 32, 107, 110, 111, 119, 32, 119, 101, 32, 111, 117, 103
, 104, 116, 32, 116, 111, 32, 226, 128, 156, 115, 117, 112, 112, 111, 114, 116, 32, 85, 110, 105,
99, 111, 100, 101, 226, 128, 157, 32, 105, 110, 32, 111, 117, 114, 32, 115, 111, 102, 116, 119, 97
, 114, 101, 32, 40, 119, 104, 97, 116, 101, 118, 101, 114, 32, 116, 104, 97, 116, 32, 109, 101, 97
, 110, 115, 226, 128, 148, 108, 105, 107, 101, 32, 117, 115, 105, 110, 103, 32, 119, 99, 104, 97,
114, 95, 116, 32, 102, 111, 114, 32, 97, 108, 108, 32, 116, 104, 101, 32, 115, 116, 114, 105, 110,
103, 115, 44, 32, 114, 105, 103, 104, 116, 63, 41, 46, 32, 66, 117, 116, 32, 85, 110, 105, 99, 111
, 100, 101, 32, 99, 97, 110, 32, 98, 101, 32, 97, 98, 115, 116, 114, 117, 115, 101, 44, 32, 97, 11
0, 100, 32, 100, 105, 118, 105, 110, 103, 32, 105, 110, 116, 111, 32, 116, 104, 101, 32, 116, 104,
111, 117, 115, 97, 110, 100, 45, 112, 97, 103, 101, 32, 85, 110, 105, 99, 111, 100, 101, 32, 83, 1
16, 97, 110, 100, 97, 114, 100, 32, 112, 108, 117, 115, 32, 105, 116, 115, 32, 100, 111, 122, 101,
110, 115, 32, 111, 102, 32, 115, 117, 112, 112, 108, 101, 109, 101, 110, 116, 97, 114, 121, 32, 97
, 110, 110, 101, 120, 101, 115, 44, 32, 114, 101, 112, 111, 114, 116, 115, 44, 32, 97, 110, 100, 3

```

```
get_stats (generic function with 1 method)
```

```
stats =
```

```
1 stats = get_stats(tokens)
```

Dict{Tuple, Integer}

► $[(101, 32) \Rightarrow 20, (240, 159) \Rightarrow 15, (105, 110) \Rightarrow 12, (226, 128) \Rightarrow 12, (32, 97) \Rightarrow 10, (97, 110) \Rightarrow 10, (115$

```
top_pair = (101, 32)
```

merge (generic function with 1 method)

```
1 function merge(ids::Vector{<: Integer}, pair::Tuple{<: Integer, <: Integer}, idx::Integer)
2     newids = Vector{Integer}() # == Integer[]
3     ix = 1
4     while ix ≤ length(ids)
5         if ix < length(ids) && ids[ix] == pair[1] && ids[ix + 1] == pair[2]
6             push!(newids, idx)
7             ix += 2
8         else
9             push!(newids, ids[ix])
10            ix += 1
11        end
12    end
13    newids
14 end
```

► [5, 6, 99, 9, 1]

```
1 merge([5, 6, 6, 7, 9, 1], (6, 7), 99)
```

```
1 begin
2     tokens_0 = merge(tokens, top_pair, 256)
3     println(tokens_0, "\nlength: ", length(tokens_0))
4 end
```

►

```
Integer[239, 188, 181, 239, 189, 142, 239, 189, 137, 239, 189, 131, 239, 189, 143, 239, 189, 13
2, 239, 189, 133, 33, 32, 240, 159, 133, 164, 240, 159, 133, 157, 240, 159, 133, 152, 240, 159, 13
3, 146, 240, 159, 133, 158, 240, 159, 133, 147, 240, 159, 133, 148, 226, 128, 189, 32, 240, 159, 1
35, 186, 226, 128, 140, 240, 159, 135, 179, 226, 128, 140, 240, 159, 135, 174, 226, 128, 140, 240,
159, 135, 168, 226, 128, 140, 240, 159, 135, 180, 226, 128, 140, 240, 159, 135, 169, 226, 128, 140
, 240, 159, 135, 170, 33, 32, 240, 159, 152, 132, 32, 84, 104, 256, 118, 101, 114, 121, 32, 110, 9
7, 109, 256, 115, 116, 114, 105, 107, 101, 115, 32, 102, 101, 97, 114, 32, 97, 110, 100, 32, 97, 1
19, 256, 105, 110, 116, 111, 32, 116, 104, 256, 104, 101, 97, 114, 116, 115, 32, 111, 102, 32, 112
, 114, 111, 103, 114, 97, 109, 109, 101, 114, 115, 32, 119, 111, 114, 108, 100, 119, 105, 100, 101
, 46, 32, 87, 256, 97, 108, 108, 32, 107, 110, 111, 119, 32, 119, 256, 111, 117, 103, 104, 116, 32
, 116, 111, 32, 226, 128, 156, 115, 117, 112, 112, 111, 114, 116, 32, 85, 110, 105, 99, 111, 100,
101, 226, 128, 157, 32, 105, 110, 32, 111, 117, 114, 32, 115, 111, 102, 116, 119, 97, 114, 256, 40
, 119, 104, 97, 116, 101, 118, 101, 114, 32, 116, 104, 97, 116, 32, 109, 101, 97, 110, 115, 226, 1
28, 148, 108, 105, 107, 256, 117, 115, 105, 110, 103, 32, 119, 99, 104, 97, 114, 95, 116, 32, 102,
111, 114, 32, 97, 108, 108, 32, 116, 104, 256, 115, 116, 114, 105, 110, 103, 115, 44, 32, 114, 105
, 103, 104, 116, 63, 41, 46, 32, 66, 117, 116, 32, 85, 110, 105, 99, 111, 100, 256, 99, 97, 110, 3
2, 98, 256, 97, 98, 115, 116, 114, 117, 115, 101, 44, 32, 97, 110, 100, 32, 100, 105, 118, 105, 11
0, 103, 32, 105, 110, 116, 111, 32, 116, 104, 256, 116, 104, 111, 117, 115, 97, 110, 100, 45, 112,
97, 103, 256, 85, 110, 105, 99, 111, 100, 256, 83, 116, 97, 110, 100, 97, 114, 100, 32, 112, 108,
117, 115, 32, 105, 116, 115, 32, 100, 111, 122, 101, 110, 115, 32, 111, 102, 32, 115, 117, 112, 11
2, 108, 101, 109, 101, 110, 116, 97, 114, 121, 32, 97, 110, 110, 101, 120, 101, 115, 44, 32, 114,
101, 112, 111, 114, 116, 115, 44, 32, 97, 110, 100, 32, 110, 111, 116, 101, 115, 32, 99, 97, 110,
32, 98, 256, 109, 111, 114, 256, 116, 104, 97, 110, 32, 97, 32, 108, 105, 116, 116, 108, 256, 105,
110, 116, 105, 109, 105, 100, 97, 116, 105, 110, 103, 46, 32, 73, 32, 100, 111, 110, 226, 128, 153
, 116, 32, 98, 108, 97, 109, 256, 112, 114, 111, 103, 114, 97, 109, 109, 101, 114, 115, 32, 102, 1
11, 114, 32, 115, 116, 105, 108, 108, 32, 102, 105, 110, 100, 105, 110, 103, 32, 116, 104, 256, 11
9, 104, 111, 108, 256, 116, 104, 105, 110, 103, 32, 109, 121, 115, 116, 101, 114, 105, 111, 117, 1
15, 44, 32, 101, 118, 101, 110, 32, 51, 48, 32, 121, 101, 97, 114, 115, 32, 97, 102, 116, 101, 114
, 32, 85, 110, 105, 99, 111, 100, 101, 226, 128, 153, 115, 32, 105, 110, 99, 101, 112, 116, 105, 1
11, 110, 46]
length: 596
```

text₁ =

"A Programmer's Introduction to Unicode March 3, 2017 · Coding · 22 Comments Unicode!         ? U\u200cN\u200c

► [65, 32, 80, 114, 111, 103, 114, 97, 109, 109, 101, 114, 226, 128, 153, 115, 32, 73, 110, 116, ... more ,111, 10

```
1 begin
2     tokens1 = (collect ∘ codeunits)(text1) # raw bytes utf-8
3     tokens1 = map(x -> Int(x), tokens1) |> collect # convert to a list of integers in range 0..255 for
4     convenience
5 end
```

```

1 begin
2     const N = 256
3     vocab_size = 276          # the desired final vocabulary size
4     num_merges = vocab_size - N
5     ids1 = collect(tokens1)    # copy so we don't destroy the original list
6
7     merges1 = Dict{Tuple{<: Integer, <: Integer}, Integer}{}
8     for ix ∈ 1:num_merges
9         stats1 = get_stats(ids1)
10        pair = argmax(stats1) # new pair
11        idx = N + ix - 1      # new token value
12        println("merging $(pair) into a new token $(idx)")
13        global ids1 = merge(ids1, pair, idx) # perform merge by replacing co-occ with the new token
14    value for pair
15        merges1[pair] = idx
16    end
17 end

```

```

merging (101, 32) into a new token 256
merging (105, 110) into a new token 257
merging (115, 32) into a new token 258
merging (116, 104) into a new token 259
merging (101, 114) into a new token 260
merging (99, 111) into a new token 261
merging (116, 32) into a new token 262
merging (226, 128) into a new token 263
merging (44, 32) into a new token 264
merging (97, 110) into a new token 265
merging (111, 114) into a new token 266
merging (100, 32) into a new token 267
merging (97, 114) into a new token 268
merging (101, 110) into a new token 269
merging (257, 103) into a new token 270
merging (261, 100) into a new token 271
merging (121, 32) into a new token 272
merging (46, 32) into a new token 273
merging (97, 108) into a new token 274
merging (259, 256) into a new token 275

```

```
Dict{Tuple{Integer, Integer}, Integer}
```

```
1 merges1 |> typeof
```

```

1 begin
2     println("tokens length: $(length(tokens1))")
3     println("ids length: $(length(ids1))")
4     @printf "compression ratio: %.2fx\n" length(tokens1) / length(ids1)
5 end

```

```

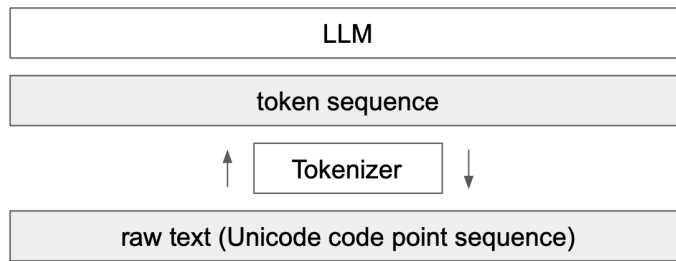
tokens length: 24597
ids length: 19438
compression ratio: 1.27x

```

Note

The Tokenizer is a completely separate, independent module from the LLM. It has its own training dataset of text (which could be different from that of the LLM), on which you train the vocabulary using the Byte Pair Encoding (BPE) algorithm. It then translates back and forth between raw text and sequences of tokens. The LLM later only ever sees the tokens and never directly deals with any text.

LLM and tokens (from Andrej Karpathy presentation)



```
1 md"""LLM and tokens (_from Andrej Karpathy presentation_) $(LocalResource("./tokenization_1.png"))"""
```

Decoding

Given a sequence of integers in the range $[0, vocab_size]$, what is the corresponding text?

decode_hof

Create a closure `decode` which given ids (list of integers), returns the corresponding string

```
1 """
2   Create a closure 'decode' which given ids (list of integers), returns the corresponding string
3   """
4   function decode_hof(merges::Dict{Tuple{Integer, Integer}, Integer})::Function
5       vocab = Dict{Integer, Vector{UInt8}}(idx => UInt8[idx] for idx ∈ 0:255)
6
7       # And now we extend the dictionary, but we need to do in order from 256..278
8       for ((p0, p1), idx) ∈ sort(merges |> collect, by=p -> p[2], rev=false)
9           vocab[idx] = vocab[p0] + vocab[p1]
10      end
11
12      function decode_fn(ids::Vector{UInt8})::String
13          collect(
14              vocab[idx][1] for idx ∈ ids
15          ) |> String
16      end
17  end
```

```
decode =
(::Main.var"workspace#230".var"#decode_fn#6"{Dict{Integer, Vector{UInt8}}}) (generic function with 1 method)
```

```
1 # vocab0
2 decode = decode_hof(merges1)
```

Encoding

Now the other way around. Given a string, what are the tokens?

encode (generic function with 2 methods)

```
1 # closure over merges1
2 function encode(text::String)::Vector{<: Integer}
3     tokens = (collect ∘ codeunits)(text)
4     while length(tokens) ≥ 2
5         stats = get_stats(tokens)
6         pair = argmin(stats)
7         pair ∉ keys(merges1) && break
8         idx = merges1[pair]
9         tokens = merge(tokens, pair, idx)
10    end
11    tokens
12 end
```

"hello world"

```
1 (decode ∘ encode)("hello world") # decode(encode("hello world", merges1), vocab0)
```



```

text2 =
"A Programmer's Introduction to Unicode March 3, 2017 · Coding · 22 Comments Unicode! 0000000? U\u200cN\u200c
1 text2 = (decode ◦ encode)(text1) # decode(encode(text1, merges1), vocab0)

```

```

1 begin
2   val_text = """Many common characters, including numerals, punctuation, and other symbols, are
   unified within the standard and are not treated as specific to any given writing system. Unicode
   encodes thousands of emoji, with the continued development thereof conducted by the Consortium as a
   part of the standard.[4] Moreover, the widespread adoption of Unicode was in large part responsible
   for the initial popularization of emoji outside of Japan. Unicode is ultimately capable of encoding
   more than 1.1 million characters."""
3
4
5   val_text2 = (decode ◦ encode)(val_text) # decode(encode(val_text, merges1), vocab0)
6   @assert val_text2 == val_text
end

```

Forced splits using regex patterns (GPT series)

Reference paper [Language Models are Unsupervised Multitask Learners](#), which uses *Byte Pair Encoding* (BPE).

```

gpt2pat = r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+\s+(?!\\S)|\\s+"
1 gpt2pat = r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+\s+(?!\\S)|\\s+"

```

```

matches = ▶ ["Hello", "'ve", " world", "123", " how", "'s", " are", " you", "!!!?" ]
1 matches = [m.match for m ∈ eachmatch(gpt2pat, "Hello've world123 how's are you!!!?")]

```

```

▶ ["for", " i", " in", " range", "(", "1", " ", " 101", "):", "\n  ", " if", " i", " %", " 3", " ==", " 0", " a
1 begin
2   example0 = ""
3   for i in range(1, 101):
4     if i % 3 == 0 and i % 5 == 0:
5       print("FizzBuzz")
6     elif i % 3 == 0:
7       print("Fizz")
8     elif i % 5 == 0:
9       print("Buzz")
10    else:
11      print(i)
12  ""
13  [m.match for m in eachmatch(gpt2pat, example0)]
14 end

```

Reference the GPT-2 encoder.py Download the vocab.bpe and encoder.json files.

```

download (generic function with 1 method)
1 function download(file_url::String)
2   local_file = split(file_url, '/') [end]
3   if !isfile(local_file)
4     println("File does not exist $(local_file), downloading it with wget...")
5     run(`wget -O $local_file $file_url`)
6   end
7 end

```

```

1 download("https://openaipublic.blob.core.windows.net/gpt-2/models/1558M/vocab.bpe")

```

```

1 download("https://openaipublic.blob.core.windows.net/gpt-2/models/1558M/encoder.json")

```

```

encoder =
▶ Dict{"ilet" ⇒ 41550, "ĠVer" ⇒ 4643, "599" ⇒ 43452, "ĠRubin" ⇒ 34599, "Ġwrestler" ⇒ 34845, "Ġsharp" ⇒ 7
1 encoder = open("./encoder.json", "r") do file
2   JSON.parse(file)
3 end

```

```
▶ [(["Ġ", "t"]), (["Ġ", "a"]), (["h", "e"]), (["i", "n"]), (["r", "e"]), (["o", "n"]), (["Ġt", "he"]), (["e", "r
```

```
1 begin
2   bpe_data = open("vocab.bpe", "r") do file
3     read(file, String)
4   end
5
6   # Process the file content into a list of tuples
7   bpe_merges = [
8     tuple(split(merge_str)) for merge_str in split(bpe_data, '\n')[2:end-1]
9   ]
10 end
```

Special tokens

50257

```
1 length(encoder) # 256 raw byte tokens + 50,000 merges + 1 special token
```

50256

```
1 encoder["<|endoftext|>"] # the only special token in use for the GPT-2 base model
```

minbpe exercise

At this point we have everything we need to build our own GPT-4 tokenizer. This is part of the `minbpe` repo, which is the solution to that exercise, and is a cleaned up version of the code presented in the original notebook (i.e python implementation).

Check original exercise and implementation: [Build your own GPT-4 Tokenizer!](#)