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

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

```julia
using Printf
```

```julia
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 `<|endoftext|>` **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 = "こにちは (Hello, Good day in Japanese)"
```

```julia
base_sentence = "こにちは (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
```

```julia
[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
```

```julia
(collect ∘ codeunits)(base_sentence)
```

```julia
begin
    # text from https://www.reedbeta.com/blog/programmers-intro-to-unicode/
    text = """Ｕｎｉｃｏｄｅ! 🅤🅝🅘🅒🅞🅓🅔‽ 🇺‌🇳‌🇮‌🇨‌🇴‌🇩‌🇪! 😄 The very name strikes fear and awe into the hearts of programmers worldwide. We all know we ought to “support Unicode” in our software (whatever that means—like using wchar_t for all the strings, right?). But Unicode can be abstruse, and diving into the thousand-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."""

    tokens = (collect ∘ codeunits)(text)          # raw bytes utf-8
    tokens = map(x -> Int(x), tokens) |> collect  # convert to a list of integers in range 0..255 for convenience

    println("---\n$(text)\nlength: $(length(text))\n---\n")
    println("$(tokens)\nlength: $(length(tokens))")
end
```

```
---
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)
```

```julia
function get_stats(ids::Vector{<: Integer})::Dict{Tuple, Integer}
    counts = Dict()
    for pair ∈ zip(ids, ids[2:end])
        counts[pair] = get(counts, pair, 0) + 1
    end
    counts
end
```

```
stats =
▸Dict((32, 84) ⇒ 1, (108, 100) ⇒ 1, (104, 105) ⇒ 1, (99, 97) ⇒ 2, (104, 101) ⇒ 6, (32, 118) ⇒ 1, (101, 4
```

```julia
stats = get_stats(tokens)
```

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

```julia
typeof(stats)
```

```
▸[(101, 32) ⇒ 20, (240, 159) ⇒ 15, (105, 110) ⇒ 12, (226, 128) ⇒ 12, (32, 97) ⇒ 10, (97, 110) ⇒ 10, (115
```

```julia
sort(stats |> collect, by=p -> p[2], rev=true)
```

```
top_pair = ▸(101, 32)
```

```julia
top_pair = argmax(stats)
```

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

```julia
function merge(ids::Vector{<: Integer}, pair::Tuple{<: Integer, <: Integer}, idx::Integer)
    newids = Vector{Integer}()  # == Integer[]
    ix = 1
    while ix ≤ length(ids)
        if ix < length(ids) && ids[ix] == pair[1] && ids[ix + 1] == pair[2]
            push!(newids, idx)
            ix += 2
        else
            push!(newids, ids[ix])
            ix += 1
        end
    end
    newids
end
```

```
▸[5, 6, 99, 9, 1]
```

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

```julia
begin
    tokens₀ = merge(tokens, top_pair, 256)
    println(tokens₀, "\nlength: ", length(tokens₀))
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! 🅤🅝🅘🅒🅞🅓🅔‽ 🇺\u200c🇳\u2
```

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

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

```julia
begin
    const N = 256
    vocab_size = 276              # the desired final vocabulary size
    num_merges = vocab_size - N
    ids₁ = collect(tokens₁)        # copy so we don't destroy the original list

    merges₁ = Dict{Tuple{<: Integer, <: Integer}, Integer}()
    for ix ∈ 1:num_merges
        stats₁ = get_stats(ids₁)
        pair = argmax(stats₁)  # new pair
        idx = N + ix - 1       # new token value
        println("merging $(pair) into a new token $(idx)")
        global ids₁ = merge(ids₁, pair, idx)  # perform merge by replacing co-occ with the new token
value for pair
        merges₁[pair] = idx
    end
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}
```

```julia
merges₁ |> typeof
```

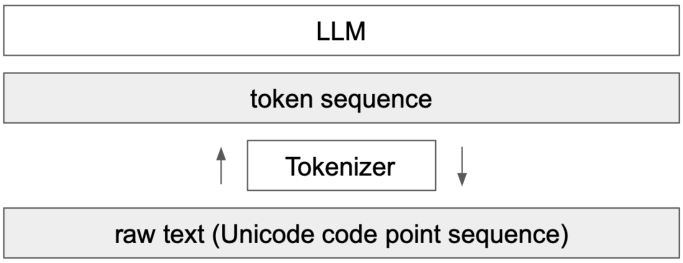
```julia
begin
    println("tokens length: $(length(tokens₁))")
    println("ids length: $(length(ids₁))")
    @printf "compression ratio: %.2fx\n" length(tokens₁) / length(ids₁)
end
```

```
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*)



```julia
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

```julia
"""
  Create a closure `decode` which given ids (list of integers), returns the corresponding string
"""
function decode_hof(merges::Dict{Tuple{Integer, Integer}, Integer})::Function
    vocab = Dict{Integer, Vector{UInt8}}(idx => UInt8[idx] for idx ∈ 0:255)

    # And now we extend the dictionary, but we need to do in order from 256..278
    for ((p₀, p₁), idx) ∈ sort(merges |> collect, by=p -> p[2], rev=false)
        vocab[idx] = vocab[p₀] + vocab[p₁]
    end

    function decode_fn(ids::Vector{UInt8})::String
        collect(
            vocab[idx][1] for idx ∈ ids
        ) |> String
    end
end
```

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

```julia
# vocab₀
decode =  decode_hof(merges₁)
```

Encoding

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

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

```julia
# closure over  merges₁
function encode(text::String)::Vector{<: Integer}
    tokens = (collect ∘ codeunits)(text)
    while length(tokens) ≥ 2
        stats = get_stats(tokens)
        pair = argmin(stats)
        pair ∉ keys(merges₁) && break
        idx = merges₁[pair]
        tokens = merge(tokens, pair, idx)
    end
    tokens
end
```

```
"hello world"
```

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

```
text₂ =
"A Programmer’s Introduction to Unicode March 3, 2017 · Coding · 22 Comments  Unicode! 🅤🅝🅘🅒🅞🅓🅔‽ 🇺\u200c🇳\u2
```

```julia
text₂ = (decode ∘ encode)(text₁)  # decode(encode(text₁, merges₁), vocab₀)
```

```julia
begin
    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."""

    val_text₂ = (decode ∘ encode)(val_text)  # decode(encode(val_text, merges₁), vocab₀)
    @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+"
```

```julia
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", "!!!?"]
```

```julia
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
```

```julia
begin
    example₀ = """
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
"""
    [m.match for m in eachmatch(gpt2pat, example₀)]
end
```

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

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

```julia
function download(file_url::String)
    local_file = split(file_url, '/')[end]
    if !isfile(local_file)
        println("File does not exist $(local_file), downloading it with wget...")
        run(`wget -O $local_file $file_url`)
    end
end
```

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

```julia
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
```

```julia
encoder = open("./encoder.json", "r") do file
    JSON.parse(file)
end
```

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

```julia
begin
    bpe_data = open("vocab.bpe", "r") do file
        read(file, String)
    end

    # Process the file content into a list of tuples
    bpe_merges = [
        tuple(split(merge_str)) for merge_str ∈ split(bpe_data, '\n')[2:end-1]
    ]
end
```

```
Vector{Tuple{Vector{SubString{String}}}} (alias for Array{Tuple{Array{SubString{String}, 1}}, 1})
```

```julia
typeof(bpe_merges)
```

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

```julia
typeof(merges₁)
```

Special tokens

```
50257
```

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

```
50256
```

```julia
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 implemnetation: Build your own GPT-4 Tokenizer!