CS639 Deep Learning for NLP

Tokenization

Junjie Hu

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Slides adapted from Percy Liang, Andrej Karpathy

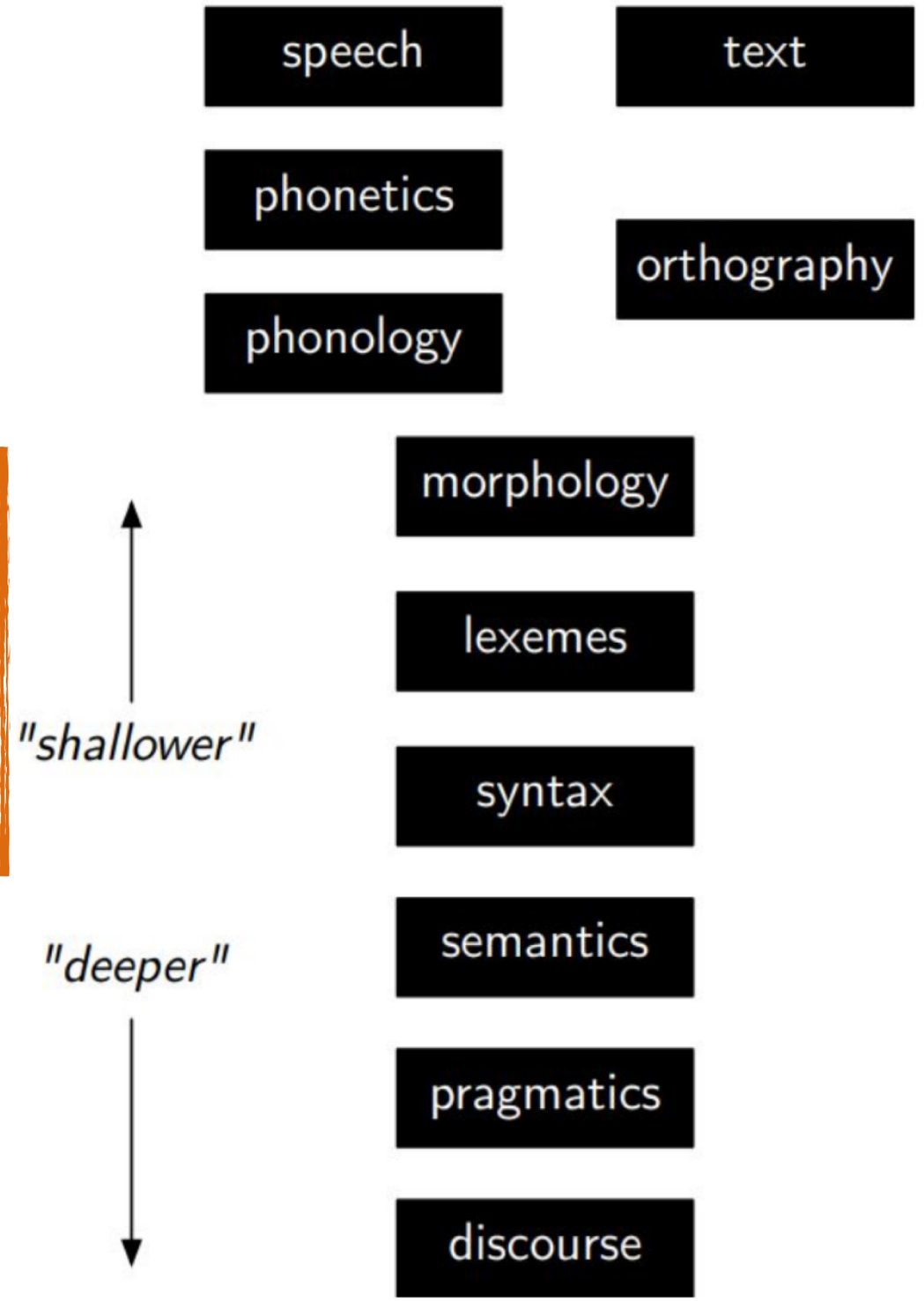
https://junjiehu.github.io/cs639-spring26/

Outline

- Morphology
- Character-based, Byte-based, Word-based Tokenization.
- **Subword** Tokenization: BPE

Levels of Linguistic Knowledge

Phonetics	The study of the sounds of human language
Phonology	The study of sound systems in human language
Morphology	The study of the formation and internal structure of words
Syntax	The study of the formation and internal structure of sentences
Semantics	The study of the meaning of sentences
Pragmatics	The study of the way sentences with their semantic meanings are used for particular communicative goals



Morphology & Word Tokenization

Morphology: Internal Structure of Words

- **Derivational morphology**: How new words are created from existing words
  - [grace]
  - [[grace]ful]
  - [un[[grace]ful]]

- **Inflectional morphology**: How features relevant to the syntactic context of a word are marked on that word.
  - This student walks.
  - These students walk.
  - These students walked.

- **Compounding**: Creating new words by combining existing words.
  - With or without space: surfboard, golf ball, blackboard

Morphemes

- **Morphemes**. Minimal pairings of form and meaning.
  - **Roots**: the "core" of a word that carries its basic meaning
    - E.g., apple, walk.
  - **Affixes** (prefixes, suffixes, infixes, and circumfixes). Morphemes that are added to a root (or a stem) to perform either derivational or inflectional functions.
    - Prefix: *un-* → negation
    - Suffix: *-s* → plural noun
    - Infix: *-ít-* → Spanish name adapted from English, e.g., Victor → Victítor
    - Circumfix: *ge- … -t* → German past participle

Morphological Parsing

- **Input**: a word
- **Output**: the word's stem(s) and features expressed by other morphemes.

Example:

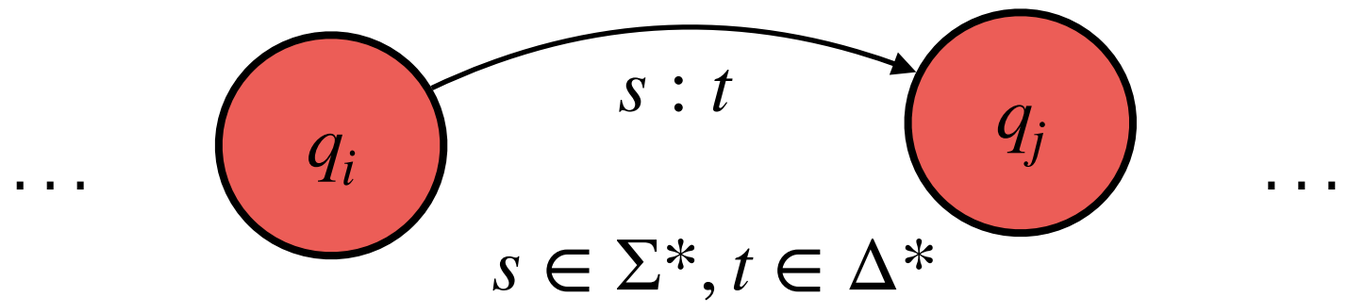
- geese → goose + N +Pl
- gooses → goose + V + 3P + Sg
- dog → {dog + N + Sg, dog + V}
- leaves → {leaf + N + Pl, leave + V + 3P + Sg}

N: Noun; Pl: Plural; V: Verb; 3P: 3rd person; Sg: singular

Finite State Transducers

- Q: a finite set of states
- $q_0 \in Q$: a special start state
- $F \subseteq Q$: a set of final states

- Σ and Δ: two finite alphabets
- Transitions:



- Encodes a set of strings that can be recognized by following paths from q_0 to some state in F

Tokenization

- Some Asian languages have no word boundary, e.g., Chinese
  - 语言学是一门关于人类语言的科学研究

- German too: Noun-noun compounds
  - *Gesundheitsversicherungsgesellschaften*
    - *Gesundheits-versicherungs-gesellschaften (health insurance companies)*

- Spanish clitics: *Dar-me-lo (To give me it)*

- Even English has issues, to a smaller degree: *Gregg and Bob's house*

Tokenization (Example)

Input raw text

```
Dr. Smith said tokenization of English is “harder than you’ve thought.”
When in New York, he paid $12.00 a day for lunch and wondered what it would
be like to work for AT&T or Google, Inc.
```

Output from Stanford Parser with Part-of-Speech tags: http://nlp.stanford.edu:8080/parser/index.jsp

```
Dr./NNP Smith/NNP said/VBD tokenization/NN of/IN English/NNP
is/VBZ ``/`` harder/JJR than/IN you/PRP 've/VBP thought/VBN ./.
''/’’
When/WRB in/IN New/NNP York/NNP ,/, he/PRP paid/VBD $/$ 12.00/CD
a/DT day/NN for/IN lunch/NN and/CC wondered/VBD what/WP it/PRP
would/MD be/VB like/JJ to/TO work/VB for/IN AT&T/NNP or/CC
Google/NNP ,/, Inc./NNP ./.
```

Tokenization approaches

- **Traditional**: Segmenting words that make sense with grammars/meanings
  - For languages with word spaces: spaces, punctuation, plus rules
  - For Chinese etc: large dictionaries, punctuation, plus rules

- **Subword-based methods**: Segmenting words to maximize processing efficiency/performance
  - Split words into subword segments *without pre-tokenization or rules.*

Traditional Tokenization

Compression Ratio

- **Definition**: the number of bytes to store a string over the number of tokens after tokenizing the string

- Also known as **bytes-per-token**. It measures the token efficiency.

```python
def get_compression_ratio(string: str, indices: list[int]) -> float:
    """Given `string` that has been tokenized into `indices`, ."""
    num_bytes = len(bytes(string, encoding="utf-8"))  # @inspect num_bytes
    num_tokens = len(indices)                         # @inspect num_tokens
    return num_bytes / num_tokens
```

Character-based Tokenization

- A Unicode string is a sequence of Unicode characters.
- Each character can be converted into a code point (integer), via the `ord` function in Python
  - ord("a") == 97
  - ord("🌍") == 127757
- It can be converted back via `chr` function in Python
  - chr(97) == “a"
  - chr(127757) == "🌍"

Character-based Tokenization

- There are approximately 150K Unicode characters.
- Problem 1: this is a very large vocabulary.
- Problem 2: many characters are quite rare (e.g., 🌍), which is inefficient use of the vocabulary.

Implementation

```python
class CharacterTokenizer(Tokenizer):
    """Represent a string as a sequence of Unicode code points."""
    def encode(self, string: str) -> list[int]:
        return list(map(ord, string))

    def decode(self, indices: list[int]) -> str:
        return "".join(map(chr, indices))
```

- string = "Hello, 🌍! 你好!"
- Indices = [72, 101, 108, 108, 111, 44, 32, 127757, 33, 32, 20320, 22909, 33]
- Compression ratio = 1.54

Byte-based Tokenization

- Unicode strings can be represented as a sequence of bytes, which can be represented by integers between 0 and 255.

- The most common Unicode encoding is “UTF-8”

- Some Unicode characters are represented by one byte:
  - e.g., bytes("a", encoding="utf-8") == b"a"

- But others take multiple bytes (1-4 bytes):
  - e.g., bytes("🌍", encoding="utf-8") == b"\xf0\x9f\x8c\x8d"

Byte-based Tokenization

- The vocabulary is nice and small: a byte can represent 256 values.

- What about the compression rate?

- The compression ratio is 1 — terrible, which means the sequences will be too long.

- Given that the context length of a Transformer is limited (since attention is quadratic), this is not looking great.

Implementation

```python
class ByteTokenizer(Tokenizer):
    """Represent a string as a sequence of bytes."""
    def encode(self, string: str) -> list[int]:
        string_bytes = string.encode("utf-8")  # @inspect string_bytes
        indices = list(map(int, string_bytes))  # @inspect indices
        return indices

    def decode(self, indices: list[int]) -> str:
        string_bytes = bytes(indices)  # @inspect string_bytes
        string = string_bytes.decode("utf-8")  # @inspect string
        return string
```

- string = "Hello, 🌍! 你好!"
- indices = [72, 101, 108, 108, 111, 44, 32, 240, 159, 140, 141, 33, 32, 228, 189, 160, 229, 165, 189, 33]
- Compression ratio = 1

Word-based tokenization

- Another approach (closer to what was done classically in NLP) is to split strings into words.

- This simple tokenization is possible for languages that **have a clear separator (e.g., whitespace)** between words (e.g., English)
  - E.g., “This is an example” —> [“This”, “is”, “an”, “example”]

- But challenging to use for some languages like Chinese that **does not use white spaces at all**
  - E.g., “这是一个例子”

Word-based tokenization

- But there are problems:
  - The number of words is huge (like for Unicode characters).
  - Many words are rare and the model won't learn much about them.
  - This doesn't obviously provide a fixed vocabulary size.
  - New words we haven't seen during training get a special **[UNK]** token, which is ugly and can mess up perplexity calculations.
  - Hard to scale up to cover multiple languages.

Sub-Word Tokenization

Subword Tokenization

- Neural systems typically use a **relatively small fixed** vocabulary
- Real world contains many words
  - New words all the time
  - For morphologically rich languages, even more so
  - But most words are rare (Zipf's Law)
- Note that rare words do not have good corpus statistics
- So, tokenize words into more frequent subword segments

Unsupervised Subword Algorithms

- Use the data to tell us how to tokenize

- Three common algorithms:
  - **Byte-Pair Encoding (BPE)** [Sennrich et al., 2016]
  - **WordPiece** [Schuster and Nakajima, 2012]
  - **Unigram language modeling tokenization** (Unigram) [Kudo, 2018]

- Learnable tokenizer:
  - Training: takes a raw training corpus and induces a vocabulary
  - Segmentation: tokenizes a raw test sentence according to the induced vocabulary

BPE: https://github.com/rsennrich/subword-nmt
SentencePiece: https://github.com/google/sentencepiece

Byte-Pair Encoding

- https://en.wikipedia.org/wiki/Byte_pair_encoding
- The BPE algorithm was introduced by Philip Gage in 1994 for data compression
- It was adapted to NLP for neural machine translation [Sennrich 2016].
- (Previously, papers had been using word-based tokenization.)
- BPE was then used by GPT-2.
- **Key idea**: *train* the tokenizer on raw text to automatically determine the vocabulary.
- **Intuition**: common sequences of characters are represented by a single token, rare sequences are represented by many tokens.

Byte-Pair Encoding

- Add a special end-of-word symbol "▁" (U+2581) or </w> at the end of each word in training corpus
- Convert words into a set of characters, create an initial vocabulary
- Iteratively merge the most frequent pair of adjacent tokens for k times

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Byte-Pair Encoding (Example)

Example — training corpus:

low low low low low lowest lowest newer newer newer newer newer newer newer wider wider wider new new

low__ low__ low__ low__ low__ lowest__ lowest__ newer__ newer__ newer__ newer__ newer__ newer__ wider__ wider__ wider__ new__ new__

corpus

```
5   l o w _
2   l o w e s t _
6   n e w e r _
3   w i d e r _
2   n e w _
```

vocabulary

```
_, d, e, i, l, n, o, r, s, t, w
```

Byte-Pair Encoding (Example)

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w
2 l o w e s t _	
6 n e w e r _	
3 w i d e r _	
2 n e w _	

Merge e r to er

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

Byte-Pair Encoding (Example)

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2 l o w e s t _	
6 n e w er _	
3 w i d er _	
2 n e w _	

Merge er _ to er_

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

Byte-Pair Encoding (Example)

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

Merge n e to ne

corpus	**vocabulary**
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t _	
6 ne w er_	
3 w i d er_	
2 ne w _	

Byte-Pair Encoding (Example)

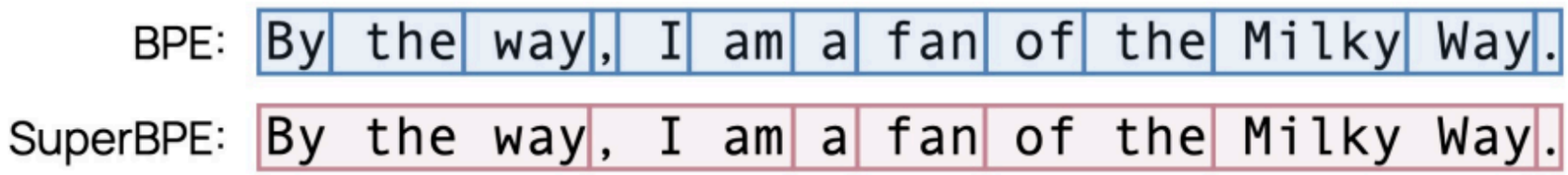
- The next merges are:

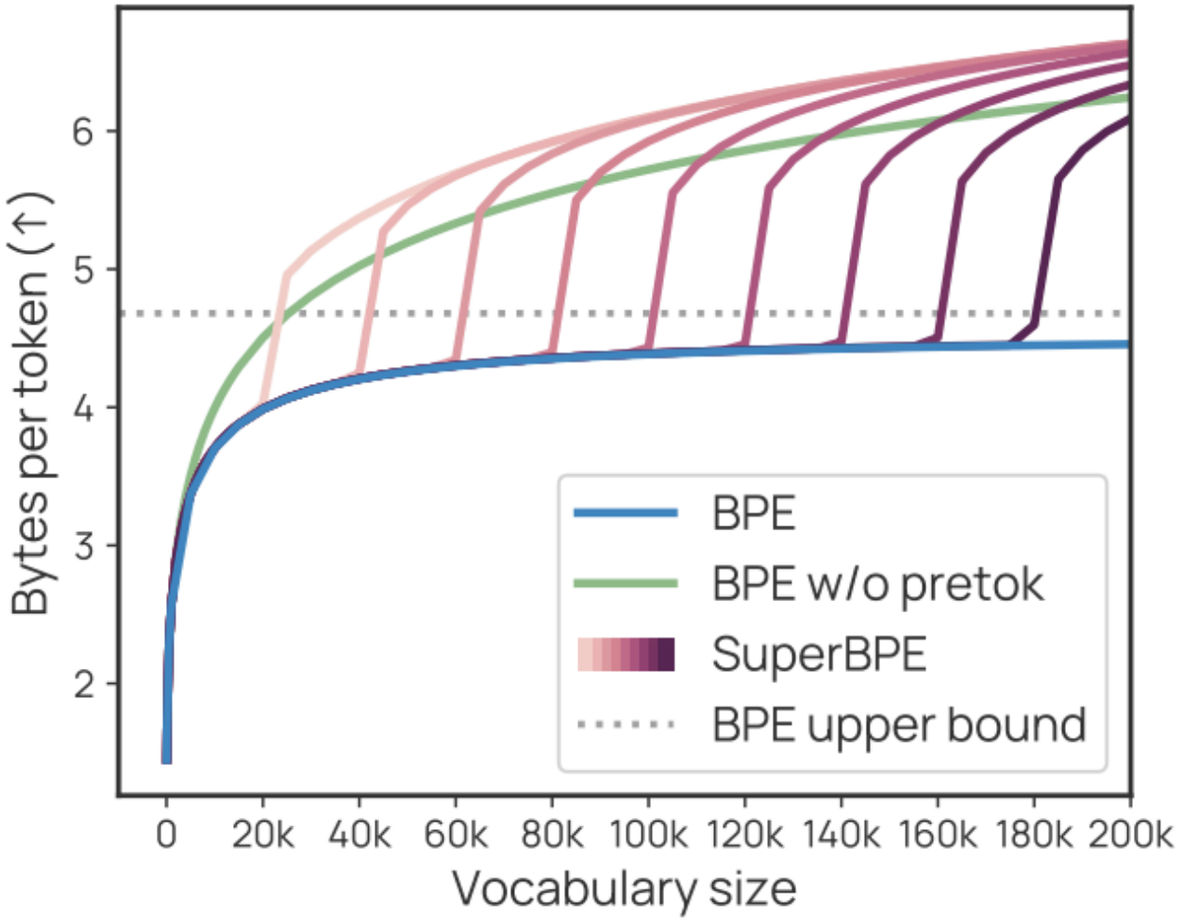
Merge	**Current Vocabulary**
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

+: Usually include frequent words,
and frequent subwords which are often morphemes, e.g., *-est* or *-er*

SuperBPE

- In addition to considering subword, a recent work considers superword (essentially phrases). It improves the token efficiency.





Tokenization in LLMs

Llama 2 Tokenizer

- The Llama 2 series, including the 405b, 70b, and 8b models, utilizes a custom tokenizer based on the SentencePiece algorithm. This tokenizer is designed for efficient processing of multiple languages and code.

- **Key Features**:
  - **Vocabulary Size**: 32,000 tokens
  - **Special Tokens**: Includes <s> (start), </s> (end), <unk> (unknown)
  - **Whitespace Handling**: Treats leading spaces as significant, important for code processing
  - **Implementation**: Written in Python, using the `**sentencepiece**` library

Llama 3 Tokenizer

- **Key Features**:
  - **Vocabulary Size:** Llama 3 uses a much larger vocabulary of **128K tokens**, a 4x increase over Llama 2's 32K token vocabulary.
  - **Token Efficiency:** The new tokenizer produces up to **15% fewer tokens** for the same input text compared to Llama 2, allowing for more text to fit within the same context window.
  - **Efficiency Gains by Content:** While generally more efficient, the Llama 3 tokenizer is particularly more efficient at encoding code, resulting in better performance for technical and programming-related tasks.
  - **Underlying Technology:** Llama 3 shifts to the **`tiktoken`** tokenizer (similar to GPT-4), which handles larger documents more efficiently and provides better support for diverse languages compared to the **`sentencepiece`** model used in Llama 2.
  - **Impact on Performance:** The increased efficiency means that for the same amount of computation, Llama 3 can process more information, leading to better overall model performance

Llama 2 vs Llama 3

Feature	Llama 2	Llama 3
Vocabulary Size	32,000	128,256
Tokenizer Type	SentencePiece	tiktoken
Efficiency (Tokens)	Baseline	~15% fewer tokens
Code Efficiency	Standard	High

GPT-4o-mini Tokenizer

- Though specific details about the GPT-4o-mini tokenizer are not publicly available, it is inferred to:

- **Key Features**:
  - **Vocabulary Size**: Approximately 50,000 tokens
  - **Tokenization Method**: Likely uses a variant of BPE
  - **Special Tokens**: Includes tokens for task separation, system messages, and control functions

Claude Sonnet 3.5 Tokenizer

- The Claude Sonnet 3.5 model by Anthropic uses a proprietary tokenizer designed to optimize the model's performance across various tasks.

- **Key Features**:
  - **Vocabulary Size**: Estimated around 48,000 tokens
  - **Tokenization Method**: Uses a variant of BPE, optimized for contextual understanding
  - **Special Tokens**: Includes tokens for different contexts and system instructions
  - **Multilingual Support**: Strong capabilities in handling multiple languages

Tokenizer online

Play around with different LLM tokenizers:

https://tokenizer.model.box/

Questions?