Tokenization & Character Encodings for NLP

Comprehensive Notes — Token Units, ASCII/Unicode, UTF-8/UTF-16/UTF-32, Hexadecimal Representation

Koorosh Asil Ghrehbaghi
K.N. Toosi University of Technology (KNTU) — B.Sc. Computer Science
Instructor: Dr. Maryam Abdolali

Compiled on: September 20, 2025

Abstract

These notes cover tokenization choices (words, whitespace, characters, subwords), character encodings (ASCII, Unicode code points), byte-oriented encodings (UTF-8, UTF-16, UTF-32), hexadecimal representation and conversion, and practical implications for NLP and LLMs (vocabulary size, memory, indexing). Each concept is explained step-by-step with worked examples, a clear algorithm for converting code points to UTF-8, and a table that groups payload bits ("xxxx / yyyy / zzzzzz / uuu") for the different UTF-8 formats.

Contents

1 Introduction: Why Tokenization and Encoding Matter

Tokenization (how we split text into units) and character encoding (how characters are represented in memory) are foundational decisions in NLP. They determine sequence length, vocabulary size, memory footprint, error modes, and how models handle multilingual text.

Definition

Tokenization: the process of splitting raw text into a sequence of discrete units (tokens) that a model or pipeline will process. Tokens may be characters, bytes, subwords, words, or multi-word expressions.

Importance & Use Cases

Choices made at tokenization and encoding time directly affect model performance, inference speed, and ability to handle rare words, symbols, or languages. In practice, modern NLP often uses subword or byte-level tokenization to balance vocabulary size and sequence length.

2 Tokenization Units: Options, Pros and Cons

We discuss four common units: characters, bytes, words (space-delimited), and subwords (BPE/WordPiece/SentencePiece).

2.1 Word (space-based) tokenization

Definition

Word tokenization splits text by whitespace and punctuation into word-like tokens ("words").

Pros:

- Intuitive and interpretable tokens.
- Shorter sequences than characters — faster for sequence models.
- Good when vocabulary is limited and text is standardized.

Cons:

- Big vocabulary (millions of word forms) — memory and OOV problems.
- Sensitive to spelling, casing, punctuation; dialects and code-switching create rare tokens.
- Not suitable for morphologically rich languages or very noisy text.

Example

Text: "I don't know."
Word tokens (simple split): ["I", "don't", "know", "."]
Depending on tokenizer, "don't" may be one token or split into "do" + "n't".

2.2 Character-level tokenization

Definition

Character tokenization treats each character (Unicode code point or code unit depending on encoding) as a token.

Pros:

- No OOV problem — can represent any text given an encoding.
- Captures morphology and can handle noisy or unseen words.

Cons:

- Much longer sequences — models must learn longer dependencies.
- May require deeper models or more compute to learn word-level semantics.

Example

"sunflowers" → [s, u, n, f, l, o, w, e, r, s]

2.3 Byte-level tokenization

Byte-level tokenizers operate on raw bytes (0–255). Example: byte-level BPE used by some GPT models.

Pros: encoding-agnostic (works on any Unicode text after UTF-8 encoding), small fixed alphabet (256 symbols), no OOV.

Cons: tokens correspond to bytes, not characters — multi-byte characters are split, sequence length increases relative to characters in some languages.

2.4 Subword tokenization (BPE, WordPiece, Unigram)

Subword tokenizers split words into frequent subword units (morphemes or common substrings). They are the pragmatic compromise used widely.

Pros: balances vocabulary size and sequence length; handles rare words by decomposing them.

Cons: requires training a tokenizer; segmentation can be unintuitive for humans; vocabulary still non-trivial in size.

Example

BPE segmentation (illustrative): "lower" → ["low", "er"]; "unbelievable" → ["un", "believ", "able"].

3 Practical trade-offs for LLMs

- Word-level: shorter sequences but enormous vocabularies and OOVs.
- Char/byte-level: minimal vocabulary (e.g., 256 bytes) but longer sequences and more compute.
- Subword: widely used (e.g., BPE) — small/medium vocab, reasonable sequence lengths.

Importance & Use Cases

For modern multilingual LLMs, byte-level or UTF-8 + subword tokenizers are popular because they are robust to arbitrary Unicode input and avoid brittle OOV behavior while keeping vocabulary manageable.

4 Character Encodings: ASCII and Unicode Overview

4.1 ASCII

ASCII (American Standard Code for Information Interchange) encodes 128 characters (0–127). Historically used for English and control codes.

Example

ASCII: ‘A’ = decimal 65 = hexadecimal 0x41 = binary 0100 0001.

4.2 Why ASCII is insufficient

ASCII cannot represent accented letters (ñ, é), CJK characters, emojis, or many symbols. Modern NLP must handle many languages and symbols.

4.3 Unicode: code points and planes

Unicode assigns each character a *code point* U+XXXX (hex). Examples:

- U+0041 ‘A’
- U+00E9 ‘é’
- U+4E16 ‘ ’ (a Chinese character)
- U+1F600 (grinning face emoji)

Unicode defines valid code points in the range 0x0000..0x10FFFF, organized in planes (Basic Multilingual Plane and others).

Definition

Code point: an abstract integer value assigned to a character, usually written as U+hex.

5 Numeric Bases and Hexadecimal Representation

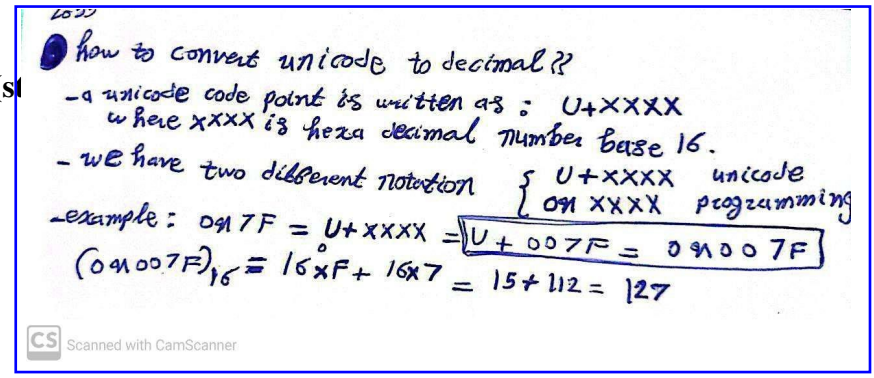
5.1 Decimal, binary, and hexadecimal

- Decimal (base 10): digits 0–9.
- Binary (base 2): digits 0–1.
- Hexadecimal (base 16): digits 0–9 and A–F (10–15).

Example

Decimal 65 in other bases:

- Binary: $65_{10} = 1000001_2$ (7 bits).
- Hex: $65_{10} = 0x41$.



5.2 Converting decimal to hexadecimal (st

Example: convert 300 to hex.

1. 300 ÷16 = 18 remainder 12 (C).
2. 18 ÷16 = 1 remainder 2.
3. 1 ÷16 = 0 remainder 1.
4. Read remainders backwards: 1 2 C ⇒ 0x12C.

6 UTF-8, UTF-16, and UTF-32: How Unicode is Encoded in Memory

6.1 UTF-8 (variable-length, 1–4 bytes)

UTF-8 encodes Unicode code points into 1 to 4 bytes. It is backward-compatible with ASCII and is dominant on the web. Rules:

1. U+0000..U+007F: 1 byte: `0xxxxxxx`.
2. U+0080..U+07FF: 2 bytes: `110xxxxx 10xxxxxx`.
3. U+0800..U+FFFF: 3 bytes: `1110xxxx 10xxxxxx 10xxxxxx`.
4. U+10000..U+10FFFF: 4 bytes: `11110uuu 10xxxxxx 10yyyyyy 10zzzzzz`.

1: 0 - 127

2: 128 - 2047

3: 2048 - 65535

4: 65536 - 1114111

Example

'A' (U+0041) in UTF-8: 0x41.
'€' (U+20AC) in UTF-8: 0xE2 0x82 0xAC.

6.2 UTF-16 (variable 2 or 4 bytes)

Uses 16-bit code units. Code points U+0000..U+FFFF take one 16-bit unit (except surrogates). Code points above U+FFFF use surrogate pairs (two 16-bit units). Example: U+1F600 → 0xD83D 0xDE00.

6.3 UTF-32 (fixed-length, 4 bytes)

Stores each code point in 4 bytes (32 bits). Advantages: constant-time indexing and simple logic. Disadvantages: 4x memory relative to 1-byte ASCII for ASCII text.

7 Algorithm: Convert a Unicode Code Point (hex) to UTF-8 bytes

This algorithm shows the exact steps you can implement and includes a table that uses grouped payload labels: `xxxx`, `yyyyyy`, `zzzzzz`, `uuu`.

Algorithm (concise steps)

1. Take a code point in hex, e.g., `U+20AC`.
2. Convert hex to binary (remove leading zeros except keep minimal necessary bits).
3. Determine UTF-8 length by numeric range:
   - $\leq$ 0x7F $\Rightarrow$ 1 byte
   - $\leq$ 0x7FF $\Rightarrow$ 2 bytes
   - $\leq$ 0xFFFF $\Rightarrow$ 3 bytes
   - $\leq$ 0x10FFFF $\Rightarrow$ 4 bytes
4. Partition the binary payload into groups to fill the template slots:
   - 2-byte: `xxxxx` (5) + `yyyyyy` (6)
   - 3-byte: `xxxx` (4) + `yyyyyy` (6) + `zzzzzz` (6)
   - 4-byte: `uuu` (3) + `xxxxxx` (6) + `yyyyyy` (6) + `zzzzzz` (6)
5. Insert each payload group into the corresponding UTF-8 byte templates and output the bytes (convert each 8-bit byte to hex).

Bit-group table (labels: `xxxx`, `yyyyyy`, `zzzzzz`, `uuu`)

Format	Template	Payload groups (names and sizes)
2-byte	`110xxxxx 10yyyyyy`	`xxxxx` (5 bits), `yyyyyy` (6 bits)
3-byte	`1110xxxx 10yyyyyy 10zzzzzz`	`xxxx` (4), `yyyyyy` (6), `zzzzzz` (6)
4-byte	`11110uuu 10xxxxxx 10yyyyyy 10zzzzzz`	`uuu` (3), `xxxxxx` (6), `yyyyyy` (6), `zzzzzz` (6)

Worked example: U+20AC (Euro sign)

1. Hex: `0x20AC`. Binary: `0010 0000 1010 1100`.
2. Range: $\leq$ 0xFFFF $\Rightarrow$ 3-byte form.
3. Group bits: `xxxx`=0010, `yyyyyy`=000010, `zzzzzz`=101100.
4. Fill template: `1110`0010 `10`000010 `10`101100.
5. Bytes (hex): `E2 82 AC`.

Worked example: U+1F600 (grinning face)

1. Hex: `0x1F600`. Binary (payload): `0001 1111 0110 0000 0000` (21 bits).
2. Range: $> 0xFFFF \Rightarrow$ 4-byte form.
3. Group bits (take lower 21 bits): `uuu`=0001 (use lower 3 bits: 001), `xxxxxx`=111101, `yyyyyy`=100000, `zzzzzz`=000000.
4. Template: `11110001 10111101 10100000 10000000`.
5. Bytes (hex): `F0 9F 98 80` (verify with code: `chr(0x1F600).encode('utf-8')`).

8 UTF-8 to code point (reverse)

1. Read the first byte; count leading 1 bits before the first 0 to determine number of bytes (0 $\rightarrow$ 1 byte, 110 $\rightarrow$ 2, 1110 $\rightarrow$ 3, 11110 $\rightarrow$ 4).
2. Mask off prefix bits from each byte (i.e., remove the `110`, `10`, `1110`, `11110` templates) and concatenate payload groups.
3. Convert the concatenated binary to hexadecimal; prepend `U+` to get the code point.

Note

If you use Python:

```
ord€('') # -> 8364
hex(ord€('')) # -> '0x20ac'€
''.encode('utf-8')# -> b'\xe2\x82\xac'
```

9 Memory, Vocabulary, and Model Implications

Embedding memory (32-bit floats) = $|V| \times d \times 4$ bytes.

Example

If $|V| = 50{,}000$ and $d = 768$:
$50{,}000 \times 768 \times 4 = 153{,}600{,}000$ bytes ≈ 146.5 MB.

- Character tokenization: tiny vocab, long sequences $\Rightarrow$ larger positional context and compute.
- Word tokenization: short sequences, huge vocab, OOV handling issues.
- Byte-level: vocabulary = 256, robust; non-Latin scripts expand input length.
- UTF-32 ("4 separated byte groups"): fixed-width indexing, simple but memory-inefficient (4 bytes per character).

Note

One-hot vectors are sparse (mostly zeros). Embeddings map token indices to dense vectors — these are stored explicitly but reduce the cost of sparse matrix operations at runtime.

10 Guidelines for Choosing Tokenization and Encoding

1. Monolingual English: UTF-8 + subword tokenization (BPE/WordPiece) is generally best.
2. Multilingual models: UTF-8 storage + byte-aware or subword tokenizers for robustness.
3. Memory-constrained: reduce |V|, lower embedding dimension, use quantization/pruning.
4. If constant-time character indexing is required and memory is available: UTF-32 (rare in practice).

11 Appendix: Quick Reference Conversions

- 'A' → U+0041 → UTF-8: 0x41
- 'é' → U+00E9 → UTF-8: 0xC3 0xA9
- '€' → U+20AC → UTF-8: 0xE2 0x82 0xAC
- '𝒜' (U+1D11E) → UTF-8: 0xF0 0x9D 0x84 0x9E

12 Further Reading and Tools

- Unicode Consortium: https://unicode.org
- RFC 3629 (UTF-8)
- Python utilities: `ord()`, `chr()`, `.encode('utf-8')`, `.decode('utf-8')`
- Books: *Unicode Explained*

Prepared by: Koorosh Asil Ghrehbaghi
Course: KNTU Computer Science (Instructor: Dr. Maryam Abdolali)