

Building an LLM from Zero

Chapter 4: Tokenization

Module 1: Foundations

Truong (Jack) Luu · jackluu.io/book

Where we are

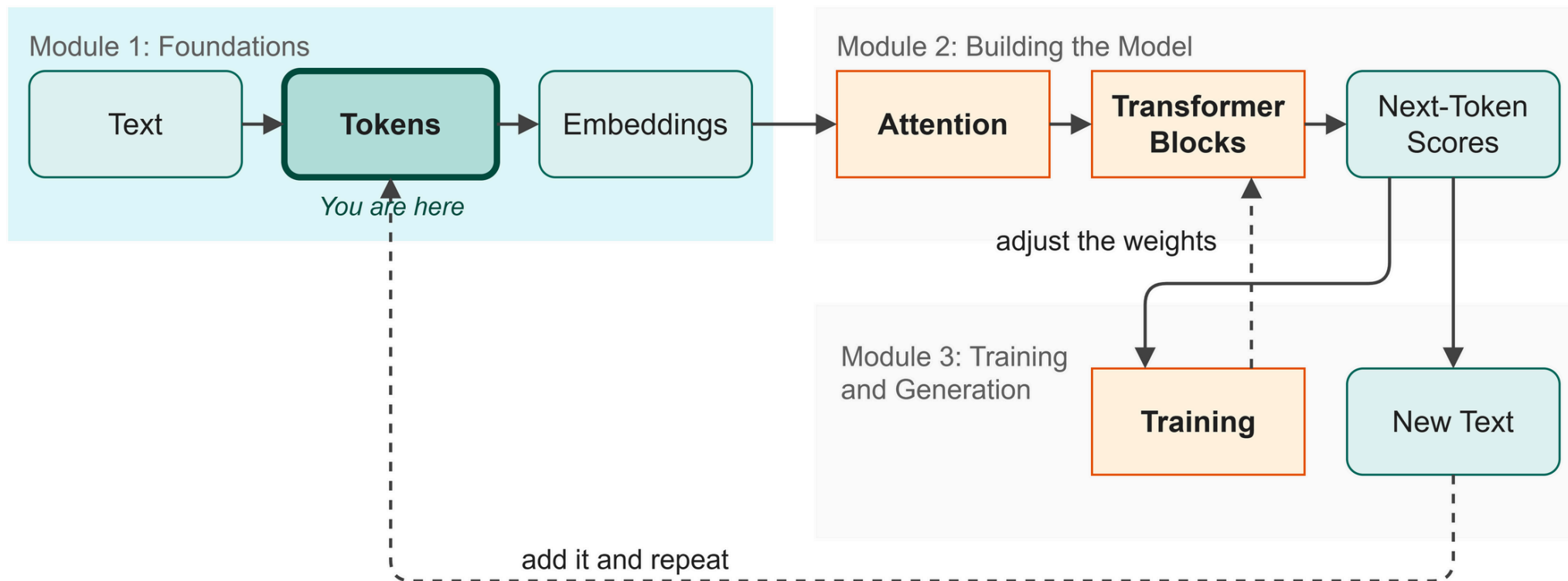


Figure 4.1: Tokenization in the big picture.

Today we build the first stage of the map: text goes in, numbers come out.

What this builds on (60-second recap)

- **Chapter 3:** a **tensor** (a multi-dimensional array of numbers) is the container every stage passes along
- Shapes read `(B, T, C)`; `@` multiplies; softmax turns scores into probabilities
- We now have somewhere to put numbers. We have no numbers yet.

Next: how does text become numbers a tensor can hold?

In this chapter you will

1. Understand why neural networks need numbers
2. Build a **character-level vocabulary** from the Shakespeare dataset
3. Write functions to **encode** text into integers and **decode** it back
4. Prepare the data for training by splitting it into two sets

Words to know (each is glossed again where it first appears):

- **Token**: one small piece of text; here, a single character
- **Tokenizer**: a function that turns text into tokens and then into numbers

Neural networks are number machines

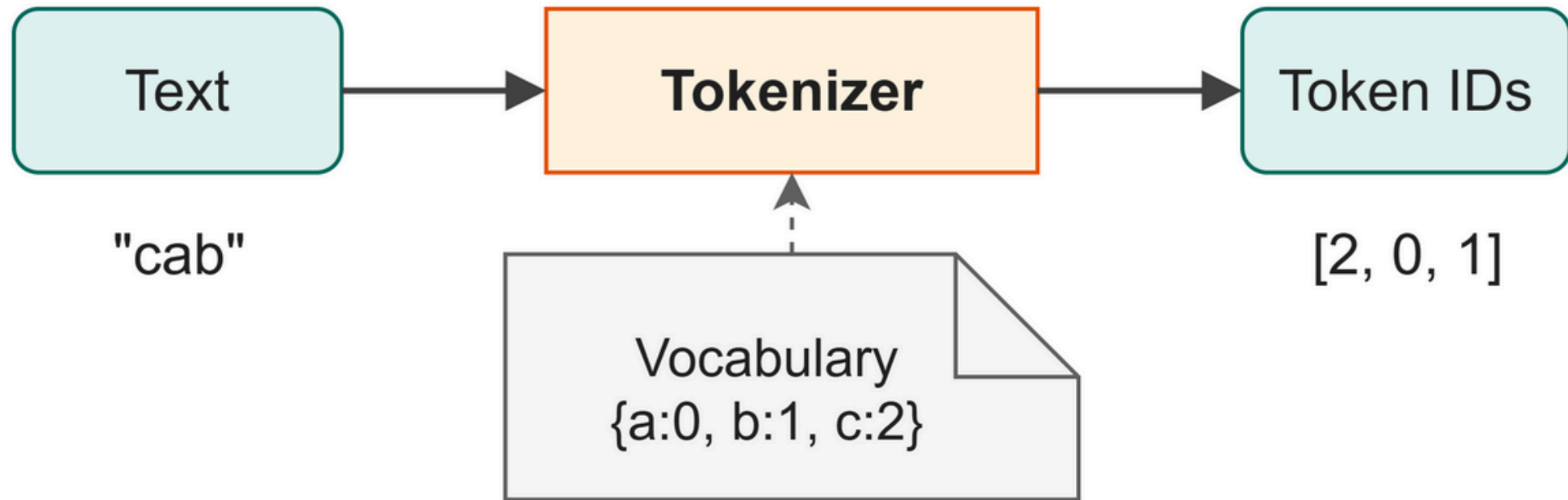


Figure 4.2: A tokenizer turns each character into a number.

A neural network cannot read. Feed it "Hello" and it has nothing to multiply and add.

Tokenization converts text into numbers, one **token** (one small piece of text) at a time.

We need a consistent rulebook: if a is 0, it must always be 0. That rulebook is the **vocabulary**.

The simplest tokenizer

We use **character-level tokenization**: every unique character gets its own integer.

- Vocabulary `{a: 0, b: 1, c: 2}` turns the word `cab` into `[2, 0, 1]`
- The Shakespeare dataset has exactly **65 unique characters**: uppercase, lowercase, spaces, punctuation
- Give each one an ID from **0 to 64** and any Shakespearean sentence becomes a list of integers

Simple on purpose. You can see the whole vocabulary on one line, and the math is the same as in a production model.

Three ways to cut text

Level	One token is	Vocabulary	Sequences
Character (this book)	one character	tiny: 65 here	long
Subword (production models)	a common chunk of letters	large	medium
Word	a whole word	massive	short

- Character-level is simple but makes sequences long
- Word-level makes sequences short but needs a massive vocabulary
- Modern assistants use **Byte-Pair Encoding** (BPE, a subword method that groups common character sequences into single tokens), a middle ground

Every choice trades vocabulary size against sequence length. We pick the one that keeps the math clean.

Hide part of the data

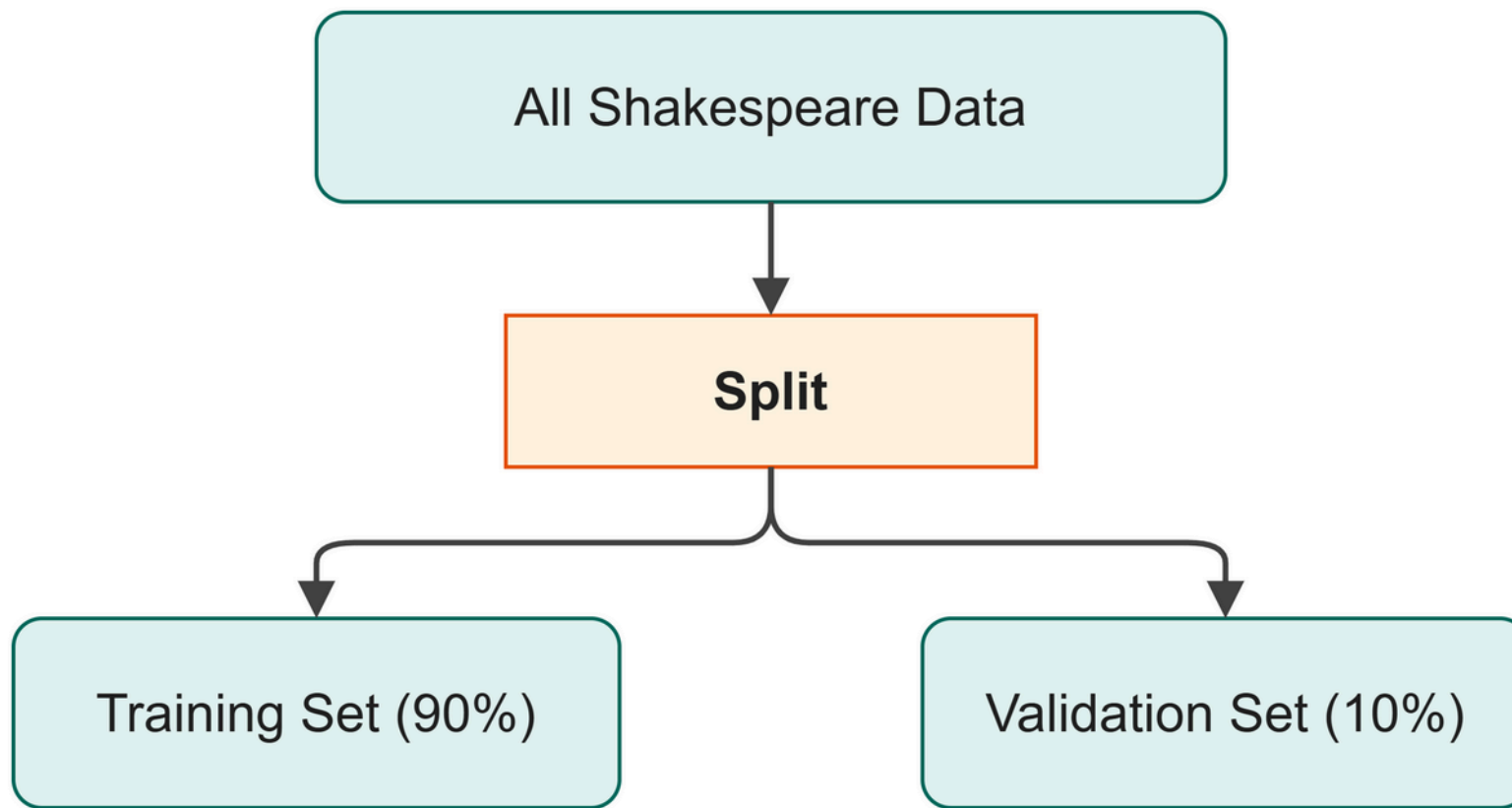


Figure 4.3: Hiding part of the data lets us test if the model actually learned.

The training and validation split

Once all of Shakespeare is one long list of numbers, we split it into two piles:

- **Training set** (90%): the data the model studies to learn patterns
- **Validation set** (10%): data we hide from the model, used to test it later

Why hide data? To know whether the model picked up patterns that hold across the text, or just memorized the passages it was shown.

Good on the training set but bad on the validation set means **overfitting**: memorizing, not learning.

Code: build the vocabulary

File: `src/ch03_tokenizer.py` (source files are numbered one lower than chapters).

```
def build_vocab(text):  
    # Find all unique characters in the text  
    chars = sorted(set(text))  
  
    # Dictionaries to translate between characters and their integer IDs  
    char_to_id = {ch: i for i, ch in enumerate(chars)}  
    id_to_char = {i: ch for i, ch in enumerate(chars)}  
  
    return chars, char_to_id, id_to_char
```

- Line 3: `set(text)` finds every unique character; `sorted()` puts them in a fixed order so the IDs stay consistent
- Lines 6 and 7: two dictionaries, one from character to ID and one back

Nine lines. That is the whole vocabulary builder.

Code: encode and decode

```
def encode(text, char_to_id):  
    # Convert a string into a list of integer IDs  
    return [char_to_id[ch] for ch in text]  
  
def decode(ids, id_to_char):  
    # Convert a list of integer IDs back into a string  
    return "".join([id_to_char[i] for i in ids])
```

- Line 3 turns a string into a list of integer IDs
- Line 7 turns the list back into a string

Encode and decode are mirror images. A round trip must give back exactly what went in.

How the script flows

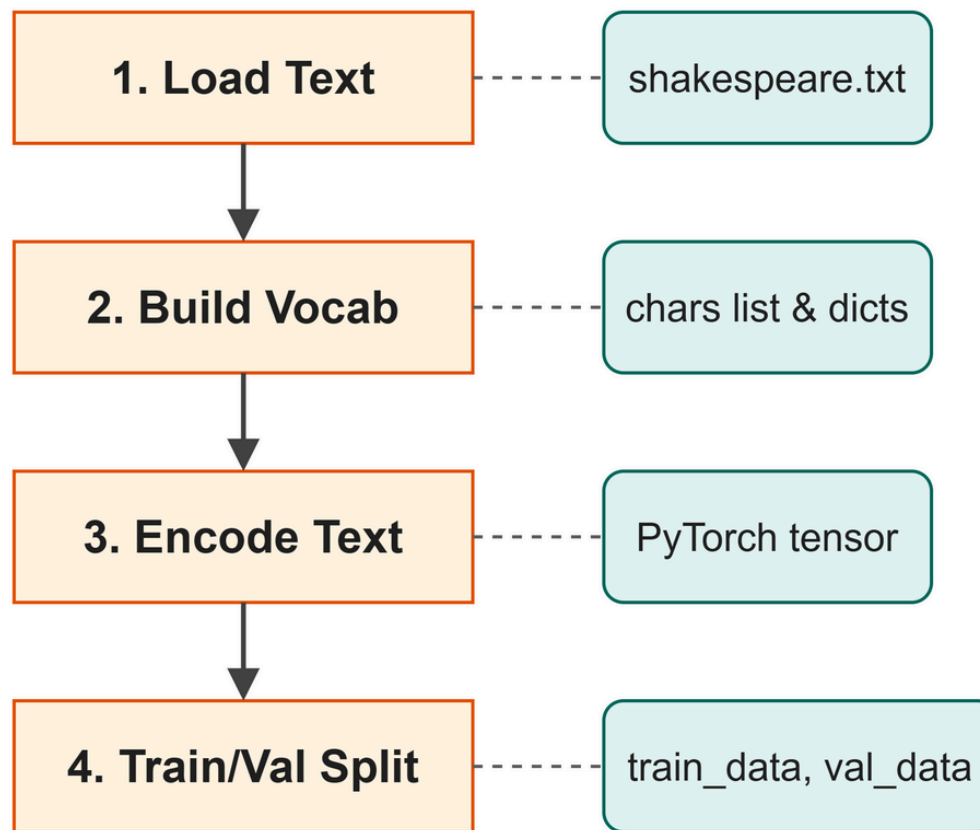


Figure 4.4: How the tokenizer processes the dataset.

Run it

```
$ python src/ch03_tokenizer.py
!$&' ,-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
...
--- Testing encode / decode ---
Original : 'Hello, World!'
Encoded   : [20, 43, 50, 50, 53, 6, 1, 35, 53, 56, 50, 42, 2]
Decoded   : 'Hello, World!'
Round-trip matches: True
```

What just happened

- The script loaded over a million characters and found exactly **65 unique characters**. The space and newline are at the start of the line.
- It translated `Hello, World!` into a list of integers, then back, and nothing was lost

Read the encoded list: `l` is 50 twice, `o` is 53 twice. Same character, same number, every time.

Run it: the split

The script then encodes the entire dataset into a PyTorch tensor and splits it:

```
Training tokens : 1,003,854
```

```
Validation tokens: 111,540
```

```
Tokenizer ready! Ready for Chapter 5.
```

- $1,003,854 + 111,540 = 1,115,394$, the character count from Chapter 1
- About 90% to train on, about 10% held back to test

The validation tokens are never shown to the model during training. They are the exam.

Try it: business phrases

Run the example script that imports the tokenizer and encodes a few business phrases:

```
$ python src/examples/ch04_tokenizer_demo.py
Demo: Encoding short business phrases
'invoice'      -> [47, 52, 60, 53, 47, 41, 43]
'URGENT!'      -> [33, 30, 19, 17, 26, 32, 2]
'Hello.'       -> [20, 43, 50, 50, 53, 8]
```

- Every letter, space and punctuation mark gets a specific number from our vocabulary
- Uppercase and lowercase are different characters, so **H** and **h** get different numbers

Try your own phrase. If it contains a character Shakespeare never used, see the next Watch Out.

In business: tokenization choices matter

For the house-style writing assistant:

- A **character-level** tokenizer is simple but makes sequences long
- A **word-level** tokenizer makes sequences short but needs a massive vocabulary
- Modern assistants use **BPE**, the middle ground: common character sequences become single tokens

We use characters here because it keeps the math clean while the model learns from its archive, our Shakespeare text.

Watch out: characters the tokenizer never saw

Our tokenizer only knows the **65 characters** it saw in the training data.

- Feed it a digit such as `1` or `2`, an emoji, or a Chinese character
- Python raises a `KeyError`: the character is not in the vocabulary

A vocabulary is a closed list. Production tokenizers solve this with subwords, so nothing is ever unknown.

Discussion

Why hide 10% of the data from the model?

In pairs, 3 minutes:

1. A student gets full marks on every practice test they have seen and fails a new one. What did they learn?
2. The model will never see the validation tokens during training. What would go wrong if it did?

Take two answers from the room.

Recap: today in 5 points

1. Neural networks process **numbers**, not text.
2. **Tokenization** translates text into numbers using a fixed **vocabulary**.
3. Our tokenizer is character-level, with **65 characters**, IDs 0 to 64.
4. `encode()` turns strings into lists of IDs; `decode()` turns them back.
5. The data is split into a **training set** (to learn) and a **validation set** (to test).

Text is now numbers. Next we give those numbers meaning.

Check your understanding

1. Why must we convert text to numbers before feeding it to a language model?
2. In our character-level tokenizer, what happens if we try to encode a character that was not in the training data?
3. What is the purpose of the validation set?
4. How many unique characters are in our vocabulary?

Next up: Chapter 5, Embeddings

- Token IDs are labels, not quantities: `Z` is not "three times" `A`
- **Embeddings** turn each ID into 128 numbers that can carry meaning, plus 128 more for its position

Before next class: run `python src/examples/ch04_tokenizer_demo.py` with a phrase of your own and bring the encoded list.

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 4. <https://jackluu.io/book/>
- Figures 4.1 to 4.4 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`).
- Terminal output: `src/ch03_tokenizer.py` and `src/examples/ch04_tokenizer_demo.py` , run on the Tiny Shakespeare text.

Further reading (from the chapter)

- Sennrich, R., Haddow, B., & Birch, A. (2015). *Neural machine translation of rare words with subword units* (arXiv:1508.07909). arXiv.

Dataset

- Tiny Shakespeare, from Andrej Karpathy's char-rnn project: <https://github.com/karpathy/char-rnn>