

# Building an LLM from Zero

## Chapter 5: Embeddings

### Module 1: Foundations

Truong (Jack) Luu · [jackluu.io/book](https://jackluu.io/book)

# Where we are

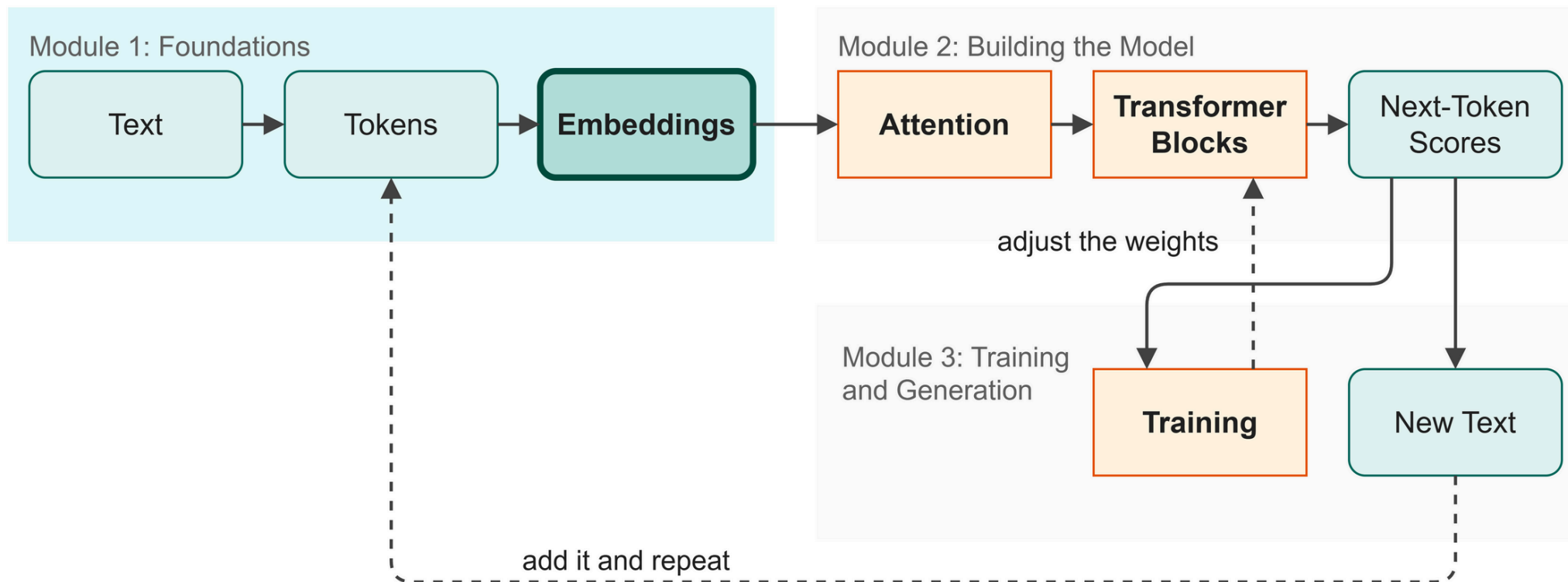


Figure 5.1: We have numbers, now we give them meaning.

Today we build the last stage of Module 1. After this, the text is fully converted into the vectors the model works on.

# What this builds on (60-second recap)

- **Chapter 4** built a **tokenizer**: 65 characters, IDs 0 to 64
- `encode()` turns text into a list of IDs; `decode()` turns it back
- The data is split 90% training, 10% validation

Next: the IDs are just labels. How does the model get meaning out of a label?

# In this chapter you will

1. Map simple token IDs to large **vectors** of numbers
2. Learn how these vectors capture meaning as **coordinates**
3. Add **position** information so the model knows the order of characters

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

- **Vector**: a list of numbers that acts as a coordinate in a high-dimensional space
- **Embedding**: a lookup table that maps a token ID to its vector

# The problem with IDs

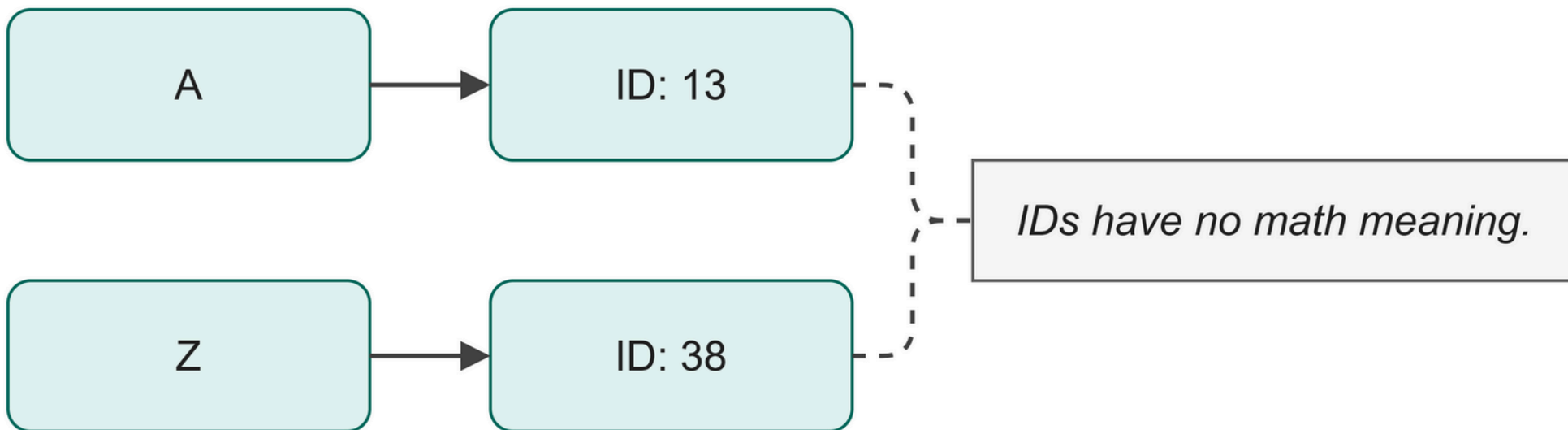


Figure 5.2: Token IDs are just arbitrary labels, not math quantities.

After tokenization, `A=13` , `B=14` , `a=39` . A network multiplies numbers, so it would see `Z` (token 38) as almost three times as big as `A` (token 13).

Employee 105 is not "five times more employee" than employee 21.

An ID is a label. The model needs something it can do math on.

# Vectors as coordinates of meaning

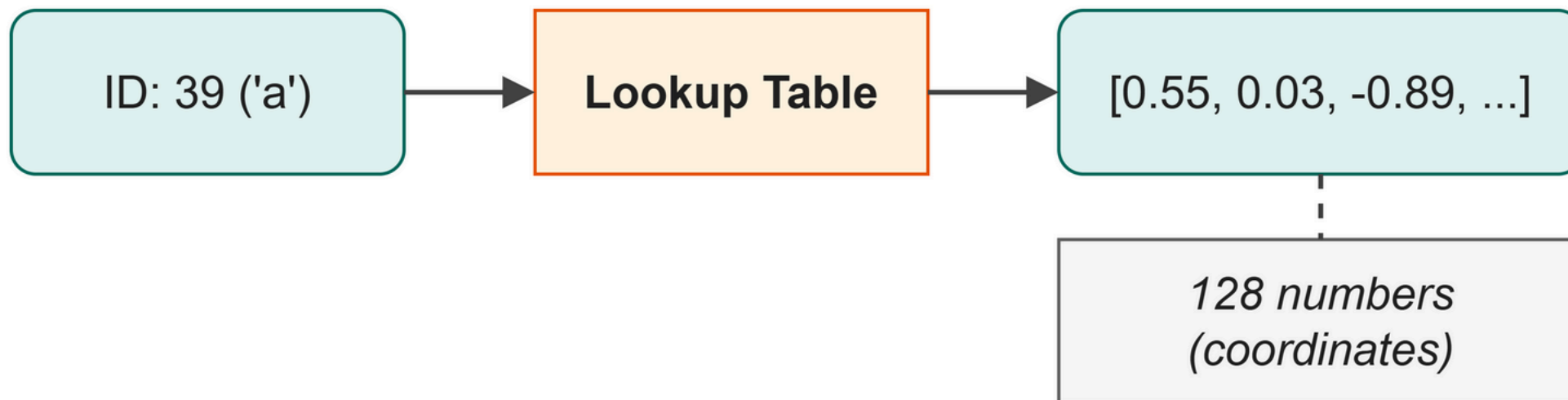


Figure 5.3: An embedding table maps each ID to a vector of numbers.

An **embedding** is a lookup table: each token ID maps to a **vector** (a list of numbers acting as a coordinate), **128 numbers** per character in our model.

Instead of 39 for a, the model sees [0.55, 0.03, -0.89, ...].

Look up, do not compute: the row for a is simply row 39 of the table.

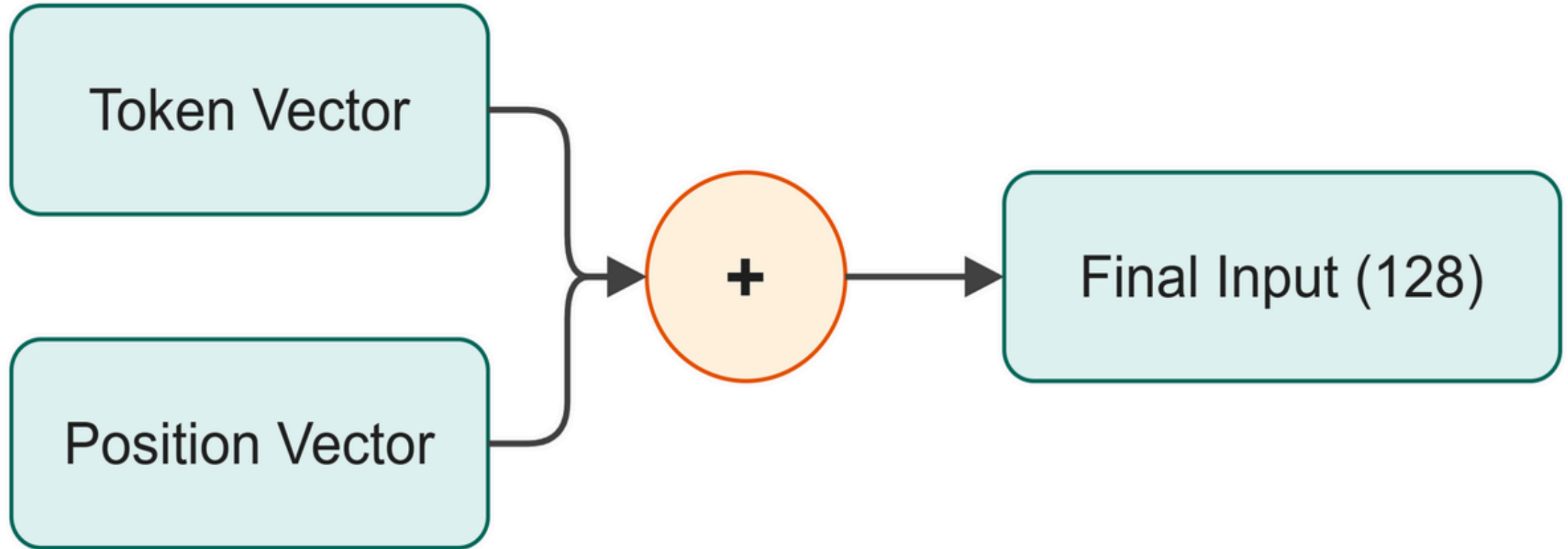
# What the vectors learn

The 128 numbers act like **coordinates** in a 128-dimensional space.

- Characters that appear in similar contexts are moved **closer together** during training
- Capital and lowercase versions of a letter behave similarly, so their vectors become similar
- The table starts random and is **learned from scratch**

Meaning is position: near each other means used alike.

# What the token is, plus where it sits



*Figure 5.4: The final representation combines what the token is with where it sits.*

# Position embeddings: where does the character sit?

- The model reads all characters at the same time. Without help, `cat` and `act` look identical: same letters
- A second lookup table, **positional embeddings**, is indexed by position (0, 1, 2, ...), not by token ID
- We **add** the token vector and the position vector, 128 numbers each

The combined 128 numbers say both "what character is this" and "where does it appear".

# Code: two lookup tables

File: `src/ch04_embeddings.py` (source files are numbered one lower than chapters).

```
token_emb = nn.Embedding(config.vocab_size, config.n_embd)
token_embeddings = token_emb(token_ids)
pos_emb = nn.Embedding(config.block_size, config.n_embd)
positions = torch.arange(T)
position_embeddings = pos_emb(positions)
x = token_embeddings + position_embeddings
```

- Line 1 creates the token table: one row per character (65 rows), 128 numbers each
- Line 3 creates the position table: one row per position
- Line 6 adds the two vectors to make the model's input `x`

Six lines. `nn.Embedding` is the lookup table; the `+` is the whole trick.

# Run it

```
$ python src/ch04_embeddings.py
--- 2. Looking up embeddings ---
Input token_ids shape : torch.Size([2, 5])
Token embeddings shape: torch.Size([2, 5, 128])
  (B=2, T=5, C=128)
...
--- 3. Positional embeddings ---
Position indices: [0, 1, 2, 3, 4]
Position embeddings shape: torch.Size([5, 128])
--- 4. Combining token + position embeddings ---
Final x shape: torch.Size([2, 5, 128])
  (B=2, T=5, C=128)
```

**What just happened:** 2 sequences of 5 token IDs went in; each ID became 128 numbers; 5 position vectors were added; the final `x` has shape `(2, 5, 128)`.

# Shape check

| Variable                         | Shape                  | Meaning                                         |
|----------------------------------|------------------------|-------------------------------------------------|
| <code>token_ids</code>           | <code>[B, T]</code>    | Batch size by time (sequence length)            |
| <code>token_embeddings</code>    | <code>[B, T, C]</code> | <code>C</code> is the embedding dimension (128) |
| <code>position_embeddings</code> | <code>[T, C]</code>    | One vector per position                         |
| <code>x</code>                   | <code>[B, T, C]</code> | The combined input for the model                |

- The position table has no batch dimension: every sequence in the batch uses the same position vectors
- PyTorch adds `[T, C]` to `[B, T, C]` by repeating it across the batch

From here on, the model's input is always `[B, T, C]`.

## Try it: change the batch, keep the tables

1. Open `src/ch04_embeddings.py`
2. Change the batch size `B` or the sequence length `T`
3. Run the script again

What the chapter says you will see: the output **shapes scale** with your change, but the **parameter counts stay exactly the same**.

The lookup tables do not care how much text you process at once. Their size depends on the vocabulary and the block size, not the batch.

# In business: a landscape of meaning

For the house-style writing assistant, embeddings turn the company archive into a landscape of meaning.

- **Token embeddings** capture the vocabulary
- **Position embeddings** give the model the order of the characters

Order is what lets the model pick up the shape of an invoice or a polite greeting.

# Watch out: index out of bounds

- If `vocab_size` is 65, the valid IDs are **0 to 64**
- Passing an ID of **65** crashes the program with an "index out of bounds" error

A lookup table has exactly as many rows as the vocabulary. Row 65 does not exist.

# Discussion

## Why 128 numbers per character, and not 1?

In pairs, 3 minutes:

1. One number per character is what the ID already was. What could a second number add? A third?
2. `cat` and `act` share three token vectors. Explain, in one sentence, how the position vectors keep them apart.

Take two answers from the room.

# Recap: today in 4 points

1. Token IDs are **arbitrary labels** and cannot be used directly for math.
2. **Embeddings** are learned lookup tables that map token IDs to vectors of 128 numbers.
3. Those vectors act as **coordinates** that capture relationships and meaning.
4. **Positional embeddings** are added so the model knows the order of the characters.

Module 1 is complete: text is now a tensor of shape  $[B, T, C]$  , ready for the model.

# Check your understanding

1. Why do neural networks struggle with raw token IDs?
2. How many numbers make up a single character's embedding vector in our model?
3. Why do we need positional embeddings in addition to token embeddings?
4. Does the size of the embedding table depend on the batch size?

## Next up: Chapter 6, Self-Attention

- Module 2 begins: the tokens start **looking at each other** to build context
- You will meet the Query, Key and Value vectors, and a mask that hides the future

**Before next class:** run `python src/ch04_embeddings.py` with `B = 3` and bring the three shapes it prints.

# References

## Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 5. <https://jackluu.io/book/>
- Figures 5.1 to 5.4 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`).
- Terminal output: `src/ch04_embeddings.py`, run on the author's laptop.

## Further reading (from the chapter)

- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient estimation of word representations in vector space* (arXiv:1301.3781). arXiv.

## Dataset

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