

Building an LLM from Zero

Chapter 6: Self-Attention (Single Head)

Module 2: Building the Model

Truong (Jack) Luu · jackluu.io/book

Where we are

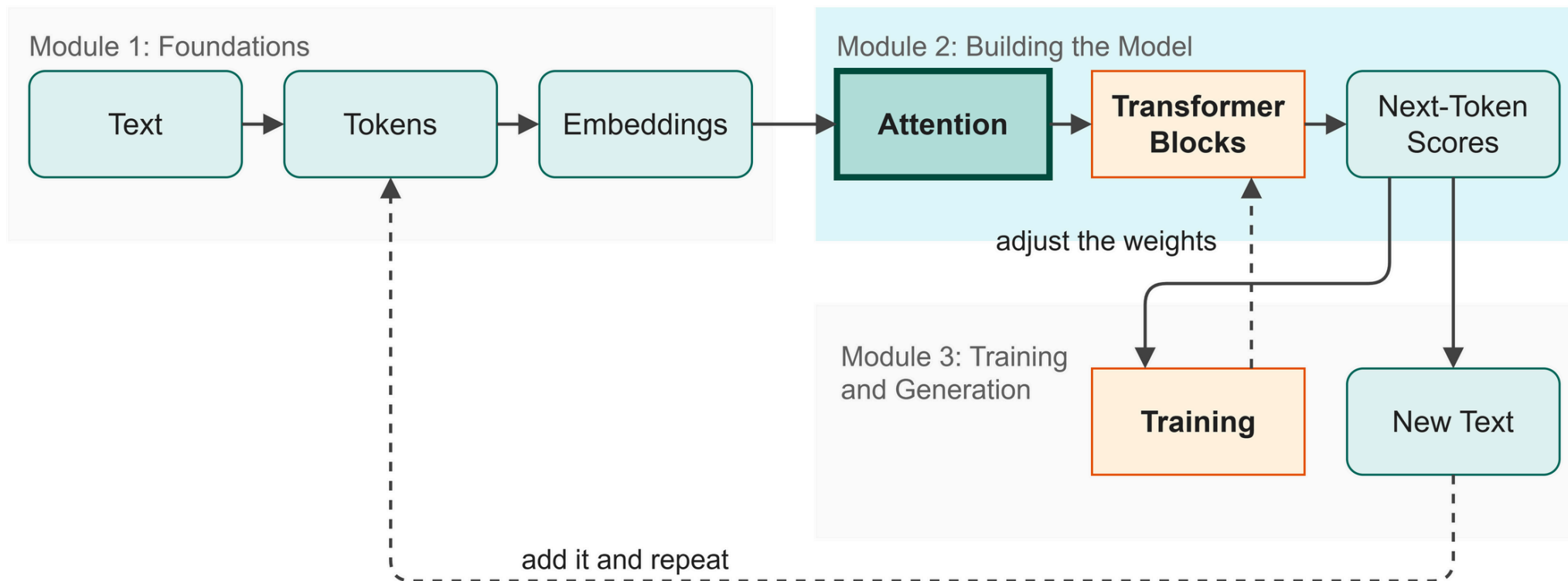


Figure 6.1: Attention builds context by looking across the sequence.

Module 2 starts here. Today the tokens start to look at each other.

What this builds on (60-second recap)

- **Chapter 5** turned each token ID into a vector of **128 numbers** through an embedding table (a learned lookup)
- A second table added **position**, so `cat` and `act` no longer look the same
- The result `x` has shape `[B, T, 128]`: batch, time, channels

Next: each token still knows only itself. How does it learn from its neighbors?

In this chapter you will

1. Learn how tokens "look" at each other to build context
2. Use the Query, Key, Value (QKV) system to find relevant information
3. See that attention is just a **weighted average** of numbers
4. Apply a **causal mask** to hide the future

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

- **Self-attention**: a mechanism where tokens evaluate every other token in the sequence to gather context
- **Query (Q)**: a vector representing what a token is looking for
- **Key (K)**: a vector representing what a token contains
- **Value (V)**: a vector holding the actual information to be shared
- **Causal mask**: a filter that prevents tokens from seeing future tokens

The problem: context matters

"The trophy didn't fit in the suitcase because **it** was too big."

To understand **it**, you connect it back to **trophy**.

- A model that reads one token at a time has no memory
- It cannot connect subjects to verbs, or adjectives to nouns
- Every token needs to look at the other tokens and decide which ones matter to its own meaning

Self-attention (each token weighs every other token) is how a token gathers context from the rest of the sequence.

Query, Key, Value: three lines of code

Self-attention gives every token three vectors. Each vector is one line of code.

The code (lines 1 to 3 of the chapter's excerpt)

```
q = self.query(x)
k = self.key(x)
v = self.value(x)
```

At a networking event

- **q**, the **Query**: what am I looking for? *A marketing expert.*
- **k**, the **Key**: what do I offer? *A badge that says "Marketing Director".*
- **v**, the **Value**: what do I actually contain? *Their advice.*

When your Query matches someone's Key, you absorb their Value. Each line multiplies the same embedding x by its own learned weights, so every token is a seeker and a source at the same time.

Attention is a weighted average

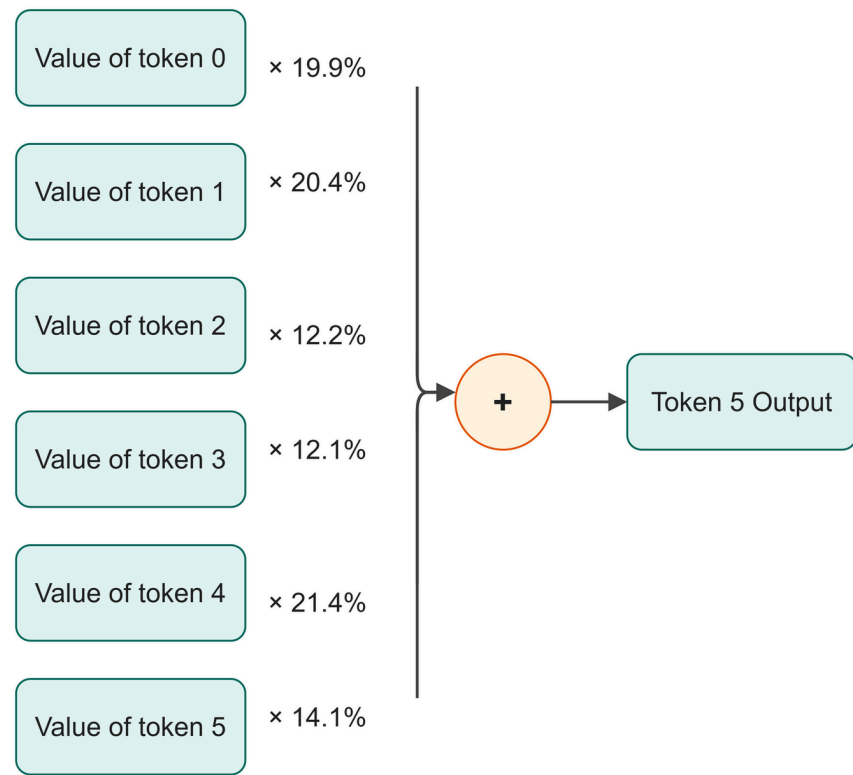


Figure 6.2: Token 5 blends information from previous tokens.

The code (line 17)

```
out = weights @ v
```

- Each box on the left is one token's **Value**, its row of `v`
- Each percentage is one **weight**, a number in `weights`
- `@` (matrix multiply, from Chapter 3) multiplies each Value by its weight and adds them up: the `+` circle

Token 5 does not pick the single best token. It mixes them all.

Score, scale, softmax: where the weights come from

Three steps turn Queries and Keys into weights. Each step is part of one line.

Step	The code	What it does
1. Score	<code>q @ k.transpose(-2, -1)</code> (line 7)	multiplies all Queries by all Keys: one score for every pair of tokens
2. Scale	<code>* scale</code> , with <code>scale = HEAD_SIZE ** -0.5</code> (lines 6 and 7)	divides by <code>sqrt(head_size)</code> (the size of one attention head, 32 here), so the numbers do not get too large
3. Softmax	<code>F.softmax(scores, dim=-1)</code> (line 13)	turns each row of scores into percentages that sum to 100% (<code>dim=-1</code> : along each row)

Score, scale, softmax. Then line 17 blends the Values with those weights. That is all attention is.

No peeking at the future

- The model processes all tokens at once, so token 5 could look at token 6
- That would be cheating: during training it would copy the answer instead of predicting it
- The **causal mask** (a triangle of negative infinities) is applied to the scores before softmax

Softmax turns $-\infty$ into exactly 0.0 . Token 5 pays 0% attention to tokens 6, 7, 8 and 9.

The causal mask in code

The **triangle**, made once and kept as `self.tril`

```
torch.tril(
    torch.ones(config.block_size, config.block_size)
)
```

The **mask**, used on every run (line 10)

```
scores = scores.masked_fill(self.tril[:T, :T] == 0, float("-inf"))
```

- `torch.tril` keeps the **lower triangle** of a square of ones: the **OK** cells
- `== 0` finds the empty cells above it, and `masked_fill` writes `-inf` there
- `block_size`: the most tokens the model reads at once (Chapter 12 covers it)

	t0	t1	t2
Token 0 sees:	OK	-inf	-inf
Token 1 sees:	OK	OK	-inf
Token 2 sees:	OK	OK	OK

Figure 6.4: A lower triangular mask ensures tokens only see the past.

Predict first: what will the run print?

```
$ python src/ch05_self_attention.py
```

The script prints how much token 5 attends to each of the ten tokens, 0 to 9.

1. Tokens 6 to 9 are in token 5's future. What weight will they get?
2. Tokens 0 to 5 share all of token 5's attention. What should their weights add up to?

Say a number before the next slide.

Run it: which line printed what

The code: `src/ch05_self_attention.py` (excerpt)

```
ch05_self_attention.py

5 # Compute attention scores and scale them to keep variance stable
6 scale = HEAD_SIZE ** -0.5
7 scores = q @ k.transpose(-2, -1) * scale
8
9 # Mask future tokens by setting their scores to negative infinity
10 scores = scores.masked_fill(self.tril[:T, :T] == 0, float("-inf")) 1
11
12 # Convert scores to probabilities and apply dropout
13 weights = F.softmax(scores, dim=-1) 2
```

What it prints

```
Terminal

$ python src/ch05_self_attention.py
Token 5 attends to tokens 0..5
(future tokens masked):

token 0: 0.199 #####
token 1: 0.204 #####
token 2: 0.122 ###
token 3: 0.121 ###
token 4: 0.214 #####
token 5: 0.141 #####
token 6: 0.000
token 7: 0.000
token 8: 0.000
token 9: 0.000
```

- 1 Line 10 writes `-inf` over the future, so tokens 6 to 9 print exactly `0.000`.
- 2 Line 13, softmax, turns the rest into weights that add up to about 1.

The weights, from a real run

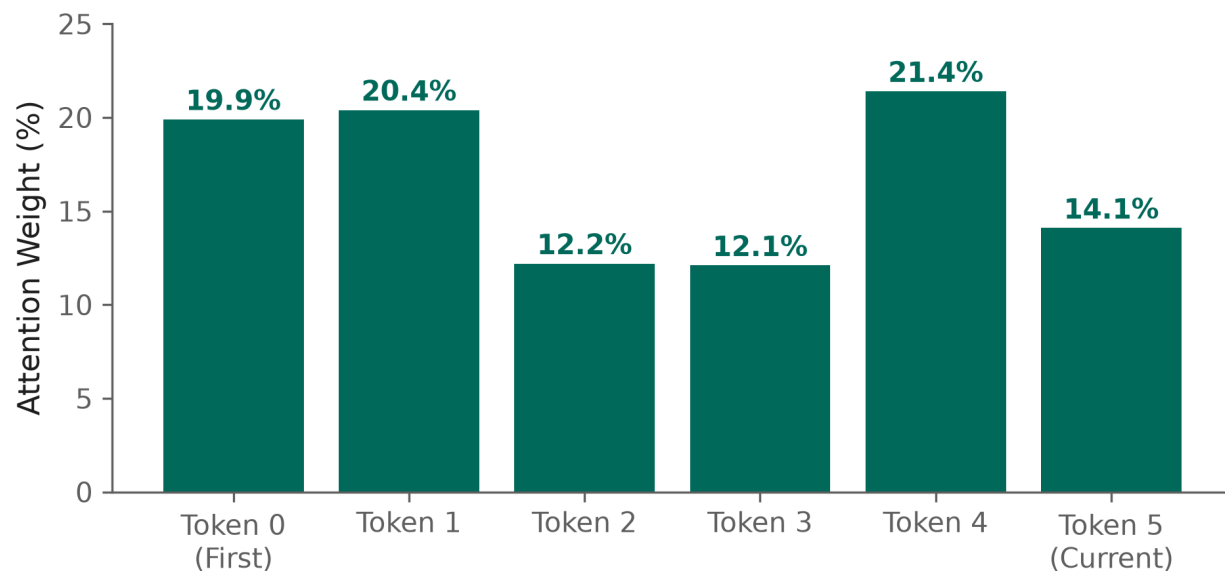


Figure 6.3: A plotted bar chart of the attention weights for token 5.

Each bar is one teal line of the run, as a percentage.

- token 0: 0.199 is the 19.9% bar
- token 4: 0.214 is the tallest bar, 21.4%
- Tokens 6 to 9 have no bar: line 10 set them to 0

The "numbers machine" builds context by blending the values of the most relevant past tokens.

How the code flows: each box is a line

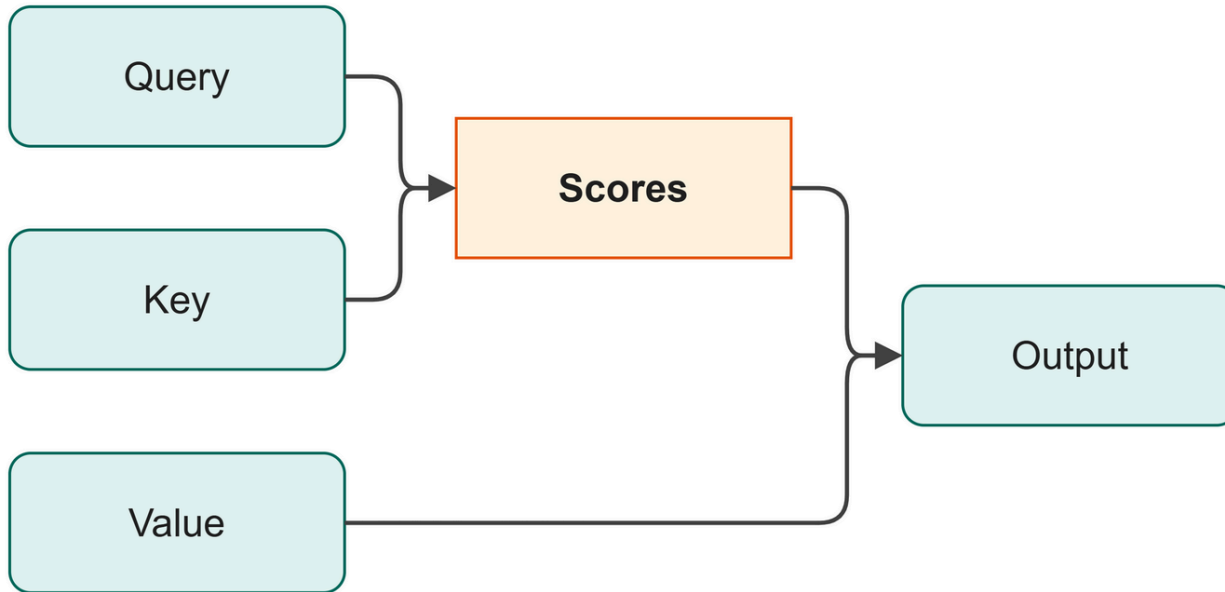


Figure 6.5: The Q, K, V vectors are created, scored, masked, and then multiplied.

- **Query, Key, Value:** lines 1 to 3
- **Scores:** lines 7 to 13, where they are scaled, masked and turned into weights
- **Output:** line 17, where the weights blend the Values

The file is

`src/ch05_self_attention.py` . It builds the `SingleHeadAttention` class. (Source files are numbered one lower than chapters.)

Shape check: what each line makes

The line	Makes	Shape	Meaning
<code>q = self.query(x)</code> , and the same for <code>k</code> and <code>v</code>	<code>q</code> , <code>k</code> , <code>v</code>	<code>[B, T, head_size]</code>	Queries, Keys and Values. <code>head_size</code> is 32.
<code>scores = q @ k.transpose(-2, -1) * scale</code>	<code>scores</code>	<code>[B, T, T]</code>	Attention scores between every pair of tokens.
<code>weights = F.softmax(scores, dim=-1)</code>	<code>weights</code>	<code>[B, T, T]</code>	The softmax probabilities, summing to 1 per row.
<code>out = weights @ v</code>	<code>out</code>	<code>[B, T, head_size]</code>	The final context-aware output vectors.

`[B, T, T]` is the giveaway: one score for every pair of tokens. That square is where the mask lives.

Try it: remove the scaling

In the demo at the bottom of the file, change `scale = HEAD_SIZE ** -0.5` to `scale = 1.0`. Run it again.

As written: `scale = HEAD_SIZE ** -0.5`

```
Terminal
$ python src/ch05_self_attention.py
...
token 0: 0.199 #####
token 1: 0.204 #####
token 2: 0.122 ###
token 3: 0.121 ###
token 4: 0.214 #####
token 5: 0.141 #####
...
```

Without the scaling: `scale = 1.0`

```
Terminal
$ python src/ch05_self_attention.py
...
token 0: 0.257 #####
token 1: 0.293 #####
token 2: 0.016
token 3: 0.015
token 4: 0.383 #####
token 5: 0.036 #
...
```

Without the scaling, the weights get more extreme: token 4 grows, and tokens 2, 3 and 5 fall close to 0.

The scaling is what keeps the model flexible and learning smoothly. Put it back before you move on.

In business: reading a whole conversation

A customer support chat, four messages long.

- A keyword search treats every word on its own
- A context-aware system uses self-attention: it connects **refund** in message 4 back to **broken screen** in message 1
- It reads the full history before it drafts a reply in the company's house style

Attention is how a system connects the question at the end to the fact at the start.

Watch out: the transpose

`k` has shape `[B, T, head_size]`. We only want to swap the **last two** dimensions.

WRONG: `scores = q @ k.T * scale`

```
Terminal
$ python src/ch05_self_attention.py
...
RuntimeError: The size of tensor a (2) must match the size
of tensor b (32) at non-singleton dimension 0
```

RIGHT: `scores = q @ k.transpose(-2, -1) * scale`

```
Terminal
$ python src/ch05_self_attention.py
--- Attention weights (what token 5 attends to) ---
...
Self-attention done! Ready for Chapter 7.
```

`.T` flips all three, so the batch size `B` (2 in this run) lines up against `head_size` (32), and the multiply crashes. `.transpose(-2, -1)` swaps only `T` and `head_size`.

Always `.transpose(-2, -1)`. The batch dimension stays where it is.

Discussion

Attention is "just a weighted average." Why does that change what a model can do?

In pairs, 3 minutes:

1. Name a sentence where the meaning of one word depends on a word far away.
2. Which tokens should get the biggest weights, and why can the causal mask still let that happen?

Take two answers from the room.

Recap: today in 4 points

1. **Self-attention** builds context by letting tokens look at other tokens.
2. The **QKV system** works like a search: Queries match Keys to retrieve Values.
3. Attention is a **weighted average** of the Values, with weights from softmax.
4. The **causal mask** sets future scores to negative infinity, so the model cannot cheat.

Next: one head is one reader. Chapter 7 runs four at once.

Check your understanding

1. What is the difference between a Query and a Key?
2. Why do we divide the attention scores by the square root of the head size?
3. What happens to a score of negative infinity when passed through softmax?
4. Why is the causal mask necessary for a language model?

Next up: Chapter 7, Multi-Head Attention

- One head is one "reader". Chapter 7 runs **four** in parallel and joins their answers
- You will check, on the trained model, whether the heads really differ

Before next class: run `python src/ch05_self_attention.py` and confirm tokens 6 to 9 print `0.000` .

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 6. <https://jackluu.io/book/>
- Figures 6.1, 6.2, 6.4 and 6.5 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`). Figure 6.3 is plotted from the script's output.
- The code and terminal windows are built by `slides/figures/build_figures.py` from the chapter's code and terminal blocks.

Further reading (from the chapter)

- Bahdanau, D., Cho, K., & Bengio, Y. (2014). *Neural machine translation by jointly learning to align and translate* (arXiv:1409.0473).
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention is all you need* (arXiv:1706.03762).

Dataset

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