

Building an LLM from Zero

Chapter 7: Multi-Head Attention

Module 2: Building the Model

Truong (Jack) Luu · jackluu.io/book

Where we are

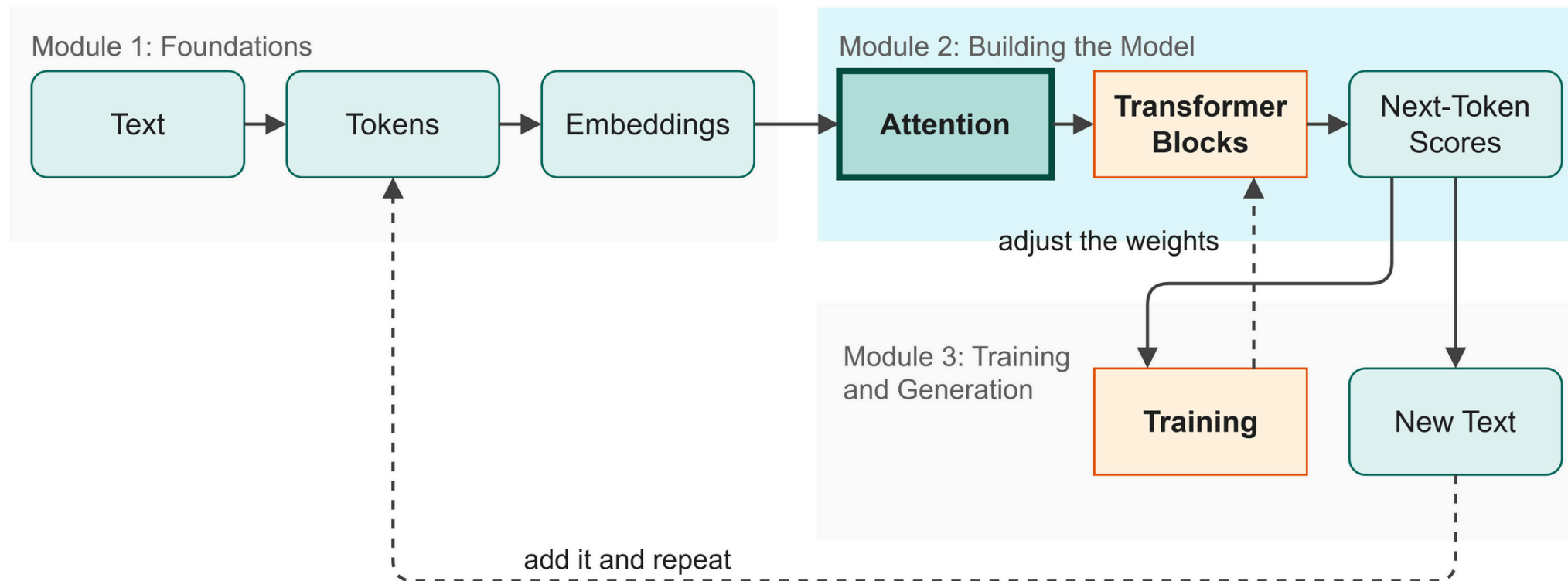


Figure 7.1: We run multiple attention heads at the same time to gather richer context.

Still in the Attention stage. Today we finish it.

What this builds on (60-second recap)

- **Chapter 6** built one attention head: Query, Key, Value, a scaled score, a causal mask, softmax
- Attention is a **weighted average** of the Values; the run printed `0.000` for every future token
- One head outputs `[B, T, 32]`: 32 numbers per token, from one "reader"

Next: one reader rarely catches everything. What if four read at once?

In this chapter you will

1. Run multiple attention heads **in parallel**
2. Understand how different heads learn to track **different patterns**
3. **Concatenate** their outputs into a single rich representation

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

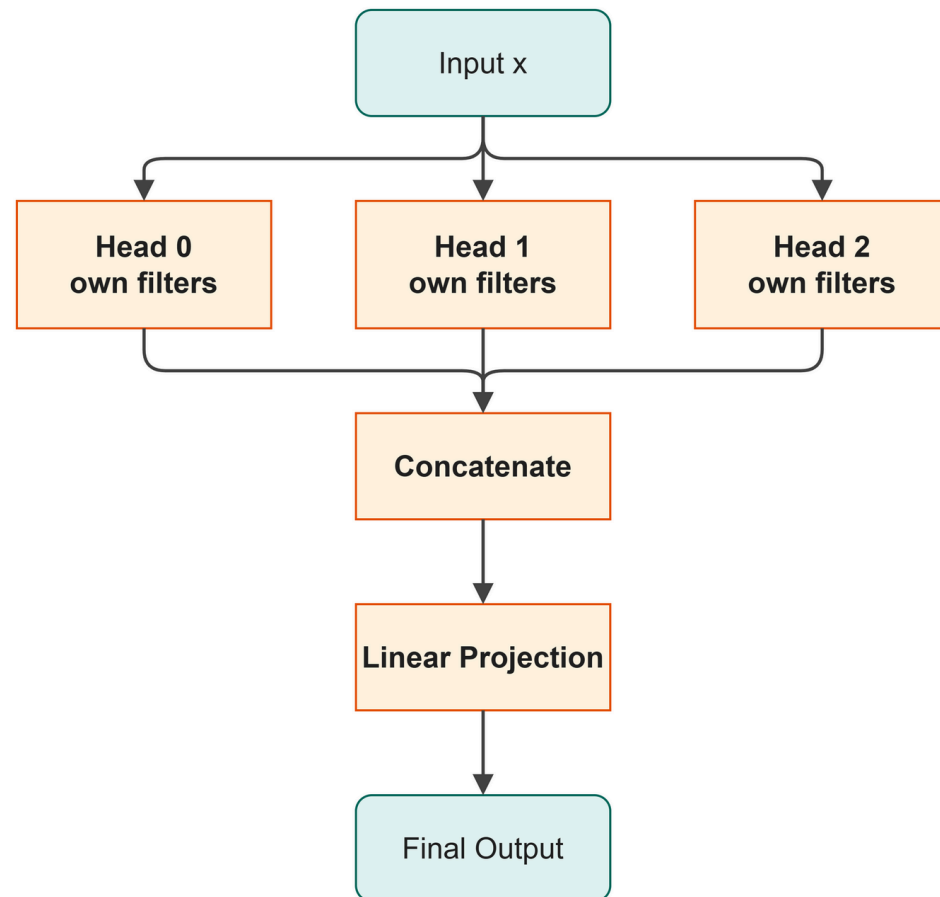
- **Multi-head attention:** running several attention operations simultaneously
- **Concatenation:** joining multiple vectors end to end to form a longer vector
- **Projection:** a linear layer that blends the concatenated outputs

One reader is not enough

Reading a sentence, you track the subject, the action and the tone at once.

A single attention head can only look for one kind of pattern at a time.

Multi-head attention (several heads run in parallel) is a team of readers. Each highlights different details, then they combine their notes.



How it works

Multi-head attention repeats the mechanism from Chapter 6:

1. Run **4** independent `SingleHeadAttention` modules on the exact same input
2. Each head produces an output of **32** numbers (`head_size`)
3. Glue (concatenate) all 4 outputs together: `4 heads × 32 numbers = 128 numbers`
4. Apply a final linear **projection** (one layer that blends them)

Each head starts with different random weights for Q, K and V, so no two heads see the text the same way, and training pushes them further apart.

Why four heads and not one big one

Four readers sound expensive. They are not.

A head's filters are sized by `head_size`, not by the full 128 channels. Split 128 across four heads and each head gets filters a quarter as wide.

```
head_size = C // n_heads

# Every head holds three filters (query, key, value), each C by head_size
four_heads = n_heads * 3 * C * head_size
one_head = 3 * C * C
```

```
$ python src/examples/ch07_head_budget.py
4 heads of 32: 49,152 numbers
1 head of 128 : 49,152 numbers
Same budget: True
```

Four points of view cost exactly the same as one. The split is free, so every model in this family uses many heads.

Do the heads really differ? The check

That is the claim. Here is the check, on the model you will train in Chapter 13.

The script loads the trained weights, feeds in a real line of Shakespeare, and prints how much attention the **last** character pays to each earlier character, one row per head.

```
scores = head.query(x) @ head.key(x).transpose(-2, -1) * head.head_size ** -0.5
weights = F.softmax(scores, dim=-1)[0, -1]      # the last character's row
```

Read the rows, not the labels. The numbers decide, not the textbook story.

Do the heads really differ? The numbers

```
$ python src/examples/ch07_heads_differ.py
```

```
Prompt: "JULIET: O Romeo"
```

Attention paid by the last character, one row per head:

	J	U	L	I	E	T	:	_	O	_	R	o	m	e	o
head 0:	0.02	0.04	0.01	0.06	0.03	0.04	0.04	0.02	0.06	0.04	0.05	0.04	0.11	0.38	0.07
head 1:	0.02	0.02	0.03	0.03	0.04	0.08	0.12	0.06	0.05	0.07	0.02	0.10	0.16	0.14	0.06
head 2:	0.04	0.07	0.04	0.06	0.05	0.11	0.04	0.06	0.09	0.07	0.07	0.06	0.09	0.09	0.06
head 3:	0.03	0.02	0.01	0.05	0.05	0.03	0.03	0.05	0.04	0.04	0.05	0.08	0.12	0.25	0.14

Sharpest focus per head: 0->'e' (0.38) 1->'m' (0.16) 2->'T' (0.11) 3->'e' (0.25)

- **Head 0** commits: 0.38 on the character just before the end, little on the rest
- **Head 3** does the same, softer: split between **e** and the final **o**
- **Head 1** spreads over the colon, the **m** and the **e**
- **Head 2** is nearly flat: largest 0.11, smallest 0.04

What the check tells you

So the heads do differ, and not in the tidy way the textbook story suggests.

- One is sharp, one is soft, one is diffuse, and one is barely committing at all
- Nobody assigned these roles, and nobody can promise that head 1 is the "grammar head"
- They are four filters that started from four random draws and were shaped by the same pressure: predict the next character

Watch out: it is tempting to read a story into each head. Sometimes a head really is that clean; often it is not, as head 2 shows. Look at the numbers before you tell the story.

The output projection: concatenate, do not average

After concatenating the 4 heads, the 128 numbers pass through one final linear layer, `proj`.

- The heads may have found redundant or conflicting information
- The projection learns to blend them: emphasize what heads 1 and 3 agree on, ignore what head 2 is unsure about

We could have **averaged** the four heads instead. That gives 32 numbers, which sounds tidy, but it throws away what we paid for: head 0's sharp focus and head 2's flat spread would cancel into a lukewarm middle.

Concatenation keeps every head's answer intact and lets the projection decide what each one is worth. Averaging decides in advance that they are all worth the same.

How the code flows

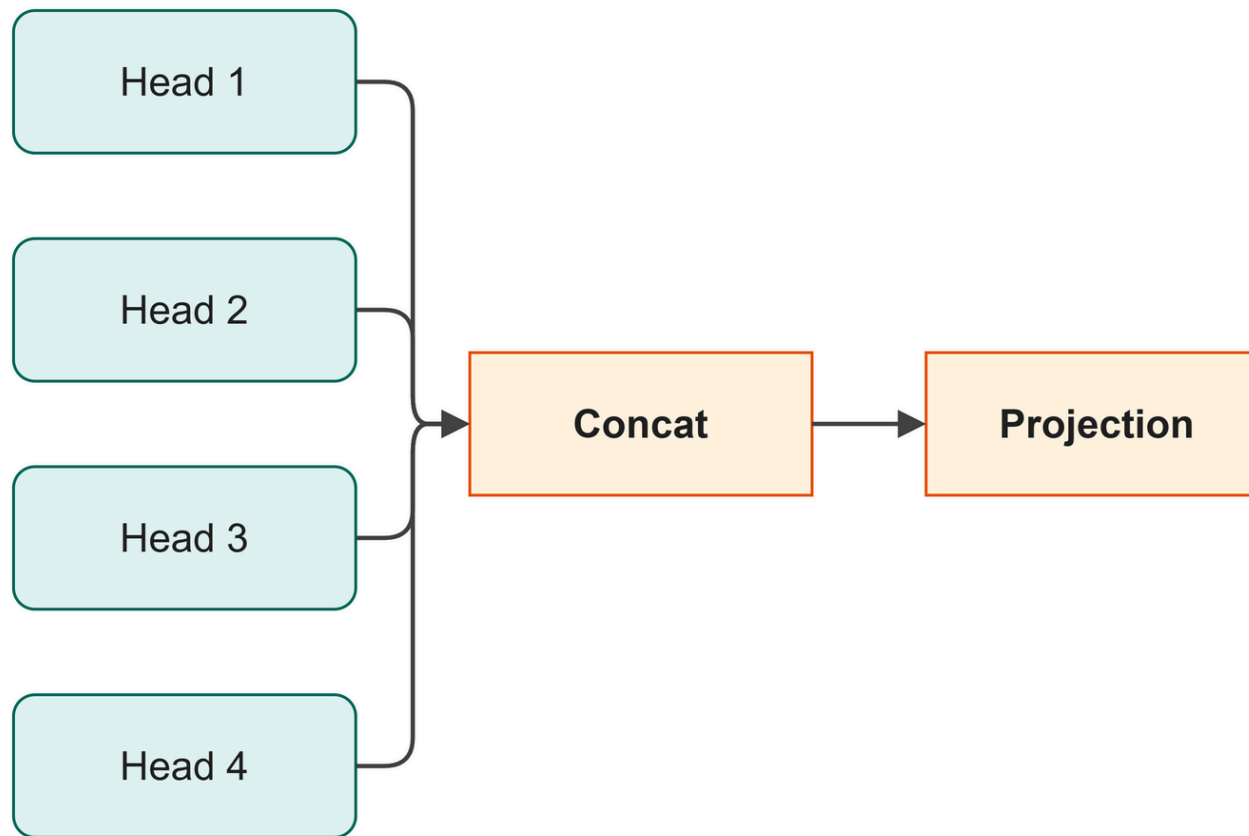


Figure 7.4: The heads run in parallel, get concatenated, and projected.

Code: four heads, joined and blended

The file is `src/ch06_multihead_attention.py`. It builds the `MultiHeadAttention` class. (Source files are numbered one lower than chapters.)

```
# Run each head in parallel
head_outputs = [h(x) for h in self.heads]

# Concatenate outputs along the last dimension
out = torch.cat(head_outputs, dim=-1)

# Apply projection and dropout
out = self.dropout(self.proj(out))
```

- Line 2 runs 4 independent attention heads on the same input; each produces 32 channels
- Line 5 concatenates them: 128 channels, the original embedding dimension
- Line 8 applies the projection and dropout so the output matches the shape of the input

Run it

```
$ python src/ch06_multihead_attention.py
--- Comparing one head vs multi-head ---
Single head output  shape: torch.Size([2, 10, 32]) (head_size=32)
Multi-head output   shape: torch.Size([2, 10, 128]) (C=128)

Multi-head output has 4x more channels - it sees 4 perspectives at once.

Multi-head attention done! Ready for Chapter 8.
```

What just happened: one head gave 32 channels per token; four heads, concatenated, gave 128. The output now has the same shape as the input, ready to stack.

Shape check

Variable	Shape	Meaning
head_outputs	$4 \times [B, T, 32]$	A list of outputs from the 4 heads.
out (after cat)	$[B, T, 128]$	The 4 outputs joined end to end.
out (final)	$[B, T, 128]$	The blended representation, ready for the next step.

In $[B, T, 128]$, out $[B, T, 128]$. That match is what lets Chapter 9 stack layers.

Try it: eight heads

1. Open `src/utils/config.py`
2. Set `n_heads = 8` instead of 4
3. Run `python src/ch06_multihead_attention.py` again

What the chapter says you will see: `head_size` drops to 16 automatically, so the concatenated size is still 128 ($8 \times 16 = 128$). More perspectives, each with less detail.

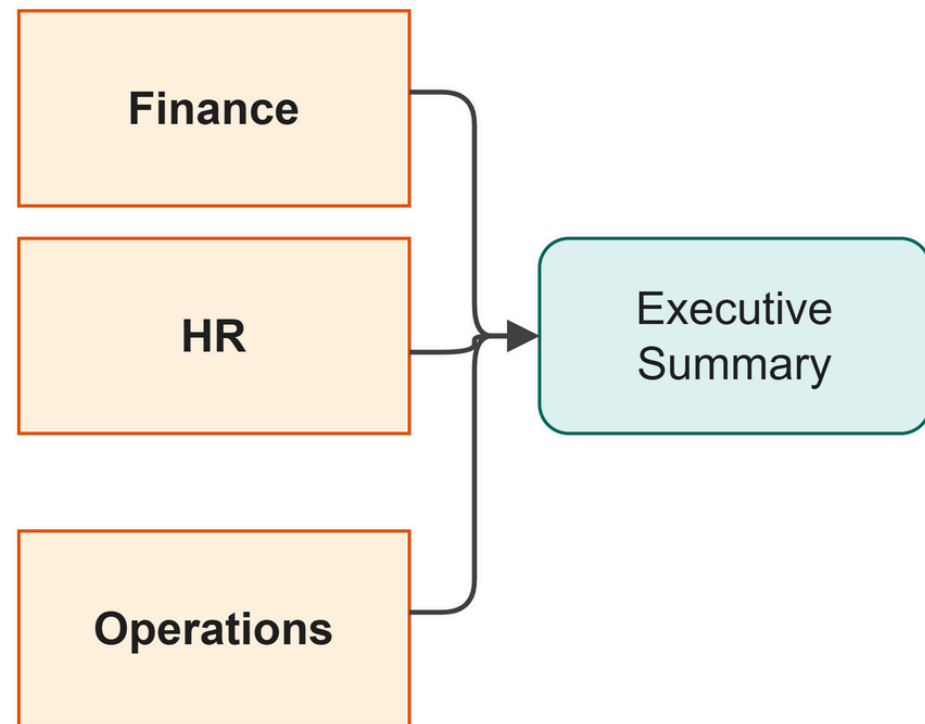
Watch out: `n_embd` must divide evenly by `n_heads`. With `n_embd = 128` and `n_heads = 5` the program crashes, because it cannot split the channels equally.

In business: several departments, one summary

An assistant that drafts emails in the house style tracks **tone**, **structure** and **vocabulary** at the same time.

Multi-head attention asks 4 "departments" to evaluate the text, then compiles one executive summary: the projection.

Different readers, different notes, one blended answer.



Discussion

Head 2 was almost flat in the run. Is a head that does very little a waste?

In pairs, 3 minutes:

1. What did the free budget slide say the four heads cost? Does a flat head change that?
2. If you could retrain, would you drop head 2 or keep it? Say what evidence you would want first.

Take two answers from the room.

Recap: today in 5 points

1. **Multi-head attention** runs several single-head modules in parallel.
2. Splitting the channels is free: four heads of 32 hold the same **49,152** numbers as one head of 128.
3. The heads do end up different, but not tidily: one sharp, one soft, one close to flat.
4. Outputs are **concatenated, not averaged**, so no head is diluted before the projection weighs it.
5. A final linear projection blends the perspectives, and the output shape `[B, T, C]` equals the input shape.

Next: attention gathered the context. Chapter 8 processes it.

Check your understanding

1. Why is one attention head not enough to understand complex text?
2. If `n_embd = 256` and `n_heads = 8`, what is the `head_size`?
3. Four heads of 32 hold the same number of weights as one head of 128. Why does splitting cost nothing?
4. Why do we concatenate the heads rather than average them?
5. In the run above, head 2's attention was almost flat. What does that tell you about the claim that each head learns its own linguistic role?
6. What is the purpose of the final projection layer?

Next up: Chapter 8, Feed-Forward and Norms

- Attention is listening to your team. Chapter 8 is going back to your desk to **think it over**
- You will build the feed-forward layer, meet the **GELU** curve and **LayerNorm**

Before next class: run `python src/examples/ch07_heads_differ.py` and find the sharpest weight in each row.

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 7. <https://jackluu.io/book/>
- Figures 7.1 to 7.4 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`).
- Terminal output: `src/examples/ch07_head_budget.py` , `src/examples/ch07_heads_differ.py` (on the trained checkpoint), `src/ch06_multihead_attention.py` .

Further reading (from the chapter)

- 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>