

Building an LLM from Zero

Chapter 10: The Full GPT Architecture

Module 2: Building the Model

Truong (Jack) Luu · jackluu.io/book

Where we are

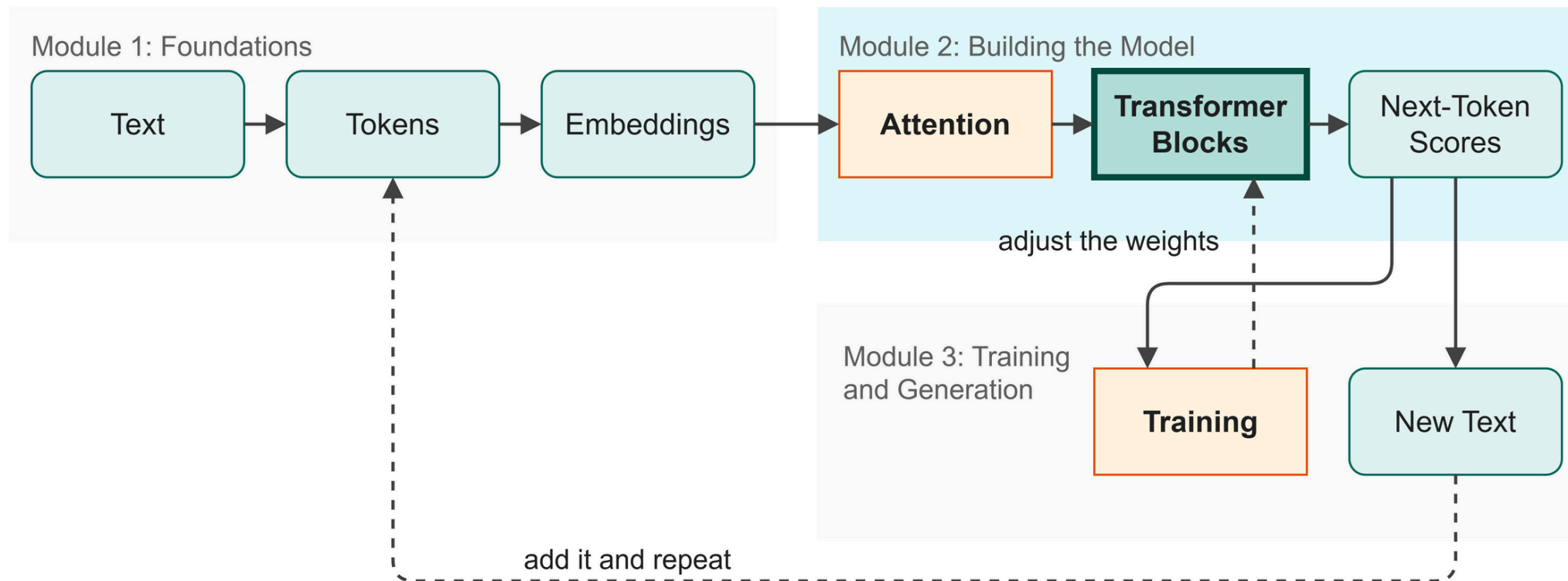


Figure 10.1: We connect everything to output a prediction.

Every part is built. Today we wire them into one model that turns text into next-token scores.

What this builds on (60-second recap)

- **Chapter 9** combined attention, feed-forward and LayerNorm into one **Transformer block**
- The $x = x + \dots$ **residual connection** (a shortcut that keeps the original signal) lets blocks stack
- One block holds **197,888** parameters; four blocks hold **791,552**
- A block keeps the shape $[B, T, 128]$, so it stacks like a brick

Next: what sits before the blocks, what sits after them, and how the whole thing becomes one model.

In this chapter you will

1. Build the **full GPT model** from end to end
2. See how data **flows through the entire pipeline**
3. Understand the final raw scores, called **logits**

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

- **LM Head**: the final linear layer that maps internal numbers back to vocabulary characters
- **Logits**: the raw scores the model outputs for each possible next token

The final assembly

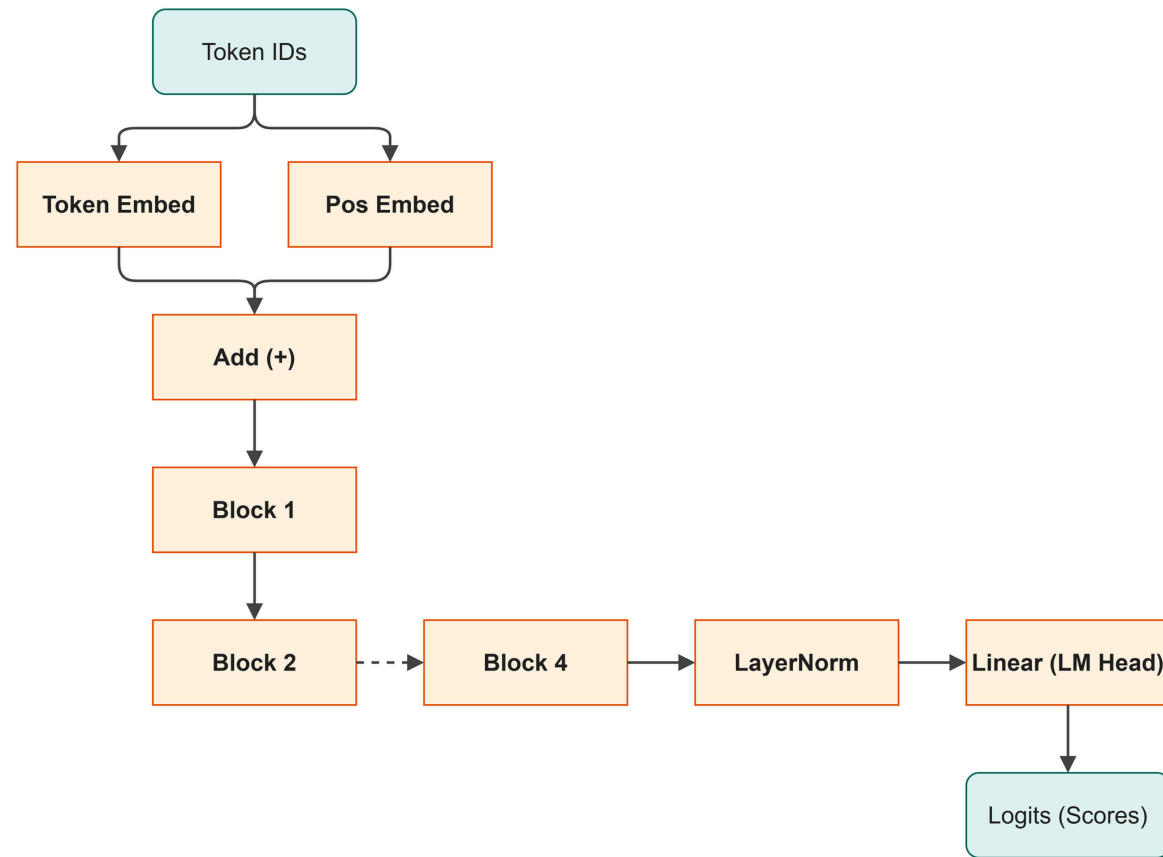


Figure 10.2: The full forward pass of the GPT model.

Read it top to bottom: the pieces you already built, in order.

Five stations on the assembly line

1. Turn text into a list of numbers (**token IDs**)
2. Look up the meaning and position vectors (**embeddings**, Chapter 5)
3. Process them through a stack of **Transformer blocks** (Chapter 9)
4. Tidy the numbers one last time (**final LayerNorm**, the step that resets numbers to a safe size)
5. Map the internal 128 numbers to the 65 possible characters (**LM head**)

Nothing new is invented here. Stations 1 to 3 are old; only 4 and 5 are new today.

Tracing one word through the model

Feed the model the word **ROMEO** (5 tokens):

Station	What happens to ROMEO
Input	The 5 token IDs [30, 27, 25, 17, 27] (R =30, O =27, M =25, E =17)
Embeddings	5 IDs become 5 vectors of 128 numbers: meaning plus position
4 blocks	The last O has looked at every earlier letter
Final LayerNorm, LM head	128 numbers become 65 scores at every position

After the blocks, the last token's 128 numbers encode something like "I am the last letter of a name from a famous play."

The LM head: back to the vocabulary

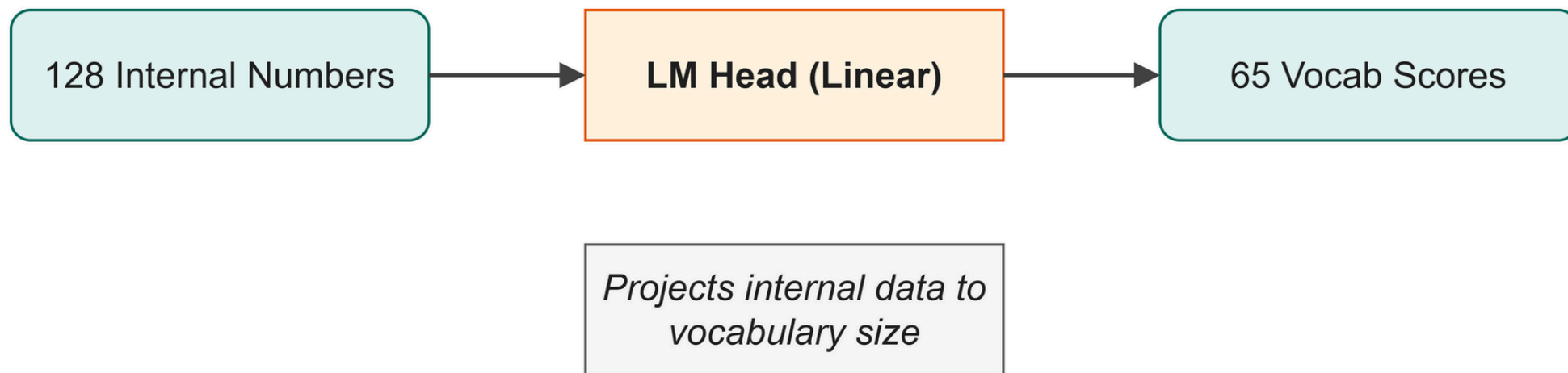


Figure 10.3: The LM Head projects internal data back to our vocabulary.

The **LM head** (Language Model head, one linear layer) turns each token's 128 numbers into **65** scores, one per character in the Shakespeare vocabulary.

Every position predicts independently. The highest score at the final position is the best guess for what follows ROME0 .

What are logits?

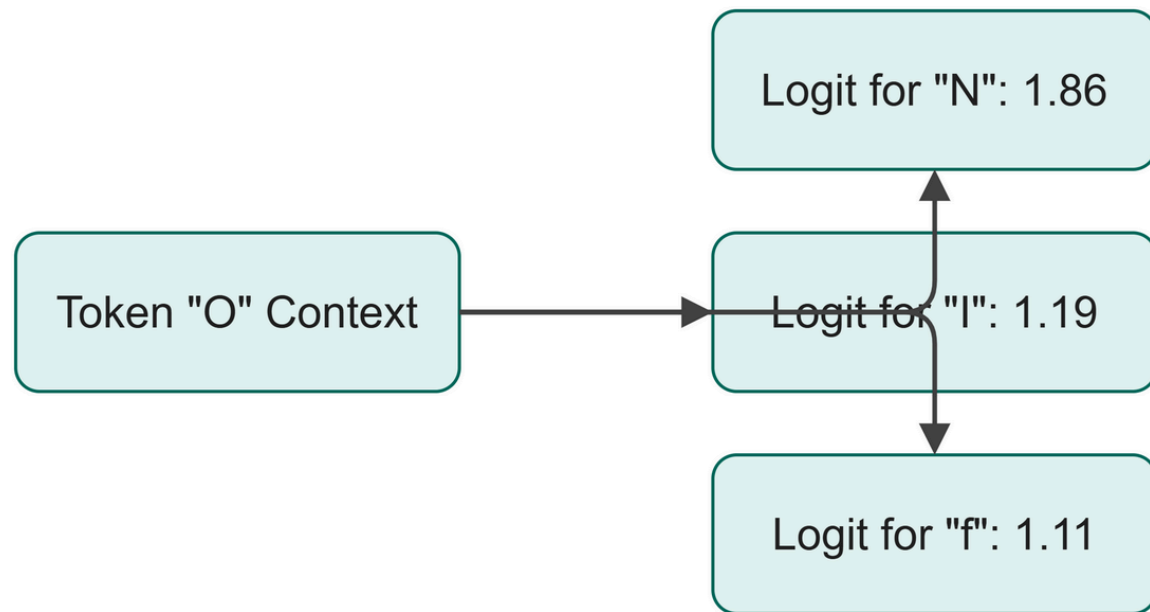


Figure 10.4: Logits are the raw scores for the next character.

Logit is the technical word for a raw score: negative, zero or very large. Logits do not add up to 1, so they are not probabilities yet.

A higher logit means more confidence in that character. Chapter 11 turns these scores into percentages.

Code: the pieces of the model

File: `src/ch09_gpt_model.py` (source files are numbered one lower than chapters)

```
# Look up table for token vectors
self.token_emb = nn.Embedding(cfg.vocab_size, cfg.n_embd)
# Look up table for position vectors
self.pos_emb = nn.Embedding(cfg.block_size, cfg.n_embd)

# A sequence of transformer blocks
self.blocks = nn.Sequential(*[
    TransformerBlock() for _ in range(cfg.n_layers)
])

# Final layer normalization and linear layer to output scores
self.ln_f = nn.LayerNorm(cfg.n_embd)
self.lm_head = nn.Linear(cfg.n_embd, cfg.vocab_size, bias=False)
```

- Lines 2 and 4 are the two lookup tables from Chapter 5; lines 7 to 9 build the blocks
- Line 13 is the LM head: 128 numbers in, 65 scores out, no bias

Code: the forward pass

The same file, the `forward` method. Data flows top to bottom:

```
tok_emb = self.token_emb(token_ids)
positions = torch.arange(T, device=token_ids.device)
pos_emb = self.pos_emb(positions)
# Combine token and position embeddings
x = tok_emb + pos_emb
# Pass through the transformer blocks
x = self.blocks(x)
# Final normalization and produce scores
x = self.ln_f(x)
logits = self.lm_head(x)
```

- Lines 1 to 5 build the input `x` exactly as Chapter 5 did; line 7 runs all four blocks in one call
- Lines 9 and 10 finish: normalize, then score every position

The five stations of the assembly line are ten lines of code.

Run it

```
$ python src/ch09_gpt_model.py
Model parameter breakdown:
  Token embedding      :      8,320
  Position embedding:  16,384
  4 Transformer blocks: 791,552
  LM head              :      8,320
  LayerNorm (final):   256
  ---
  TOTAL                :    824,832
Input shape: torch.Size([2, 10])  (B, T)
Output shape: torch.Size([2, 10, 65]) (B, T, vocab_size)
...
```

What just happened

- The complete model has **824,832** parameters (weights)
- The original GPT-2 Small had 117 million; ours is the same architecture, scaled down
- For 2 sequences of 10 tokens in, the model gives 65 scores at every position

Where the 824,832 parameters live

Piece	Count from the run	Why that size
Token embedding	8,320	65 characters × 128 numbers
Position embedding	16,384	128 positions × 128 numbers
4 Transformer blocks	791,552	4 × 197,888 (Chapter 9)
LM head	8,320	128 numbers × 65 characters
Final LayerNorm	256	2 × 128
Total	824,832	

The blocks hold almost all of the model. The head and the two lookup tables are small.

Shape check

Tensor	Shape	Meaning
Input token IDs	[Batch, Time]	Which character sits at each position
Output logits	[Batch, Time, 65]	65 scores for the next character, at every position

One number per character goes in. Sixty-five scores per character come out.

Try it: make the model deeper

1. Open `src/ch09_gpt_model.py`
2. Set `config.n_layers = 6` (just to test)
3. Run the script again

The chapter says you will see the parameter count increase. The blocks contain the vast majority of the model's "brain".

Depth is one number in the config. The rest of the pipeline does not change.

In business: a pipeline from modular parts

For the house-style writing assistant, the full model is a production pipeline:

- A **data reader**: the embeddings
- An **analysis engine**: the Transformer blocks
- An **output formatter**: the LM head

Because it is modular, you can upgrade the engine (add more layers) without rewriting the rest.

Watch out: the block size limit

The model can read at most `block_size` tokens at once (128 in this book). The code checks it before anything else runs:

```
assert T <= self.cfg.block_size, \
    f"Sequence length {T} exceeds block_size " \
    f"{self.cfg.block_size}"
```

- The position table has exactly 128 rows, one per position, so a longer input asks for a row that does not exist
- The check stops the program with this message instead of a confusing error later

Position embeddings set a hard ceiling on how much text the model can see at once.

Discussion

The model scores every position, not only the last one.

In pairs, 3 minutes:

1. For `R0ME0` , the model makes 5 predictions. What is each one predicting?
2. Why might computing all 5 be useful, if we only want the character after `R0ME0` ?

Take two answers from the room.

Recap: today in 4 points

1. The full model stacks **embeddings**, **Transformer blocks**, a final LayerNorm and an **LM head**.
2. The LM head is a linear layer that projects 128 internal numbers to the 65 vocabulary characters.
3. The model outputs **logits**: raw, unnormalized scores for the next token.
4. The complete model has **824,832** parameters, and the blocks hold almost all of them.

The model is assembled. Right now it only guesses at random. Next we give it a goal.

Check your understanding

1. If the input sequence has 10 tokens, how many predictions does the model make?
2. Why does the LM head output exactly 65 numbers per token?
3. What is a logit?

Bonus: which single line of the config would you change to make the model read more characters at once, and which parameter count would grow?

Next up: Chapter 11, Causal Language Modeling

- The model has an output. Next it gets a **goal**: predict the next token from the past only
- You will build input and target pairs and measure the model's **surprise** with cross-entropy loss

Before next class: run `python src/ch09_gpt_model.py` and write down the total parameter count and the output shape.

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 10. <https://jackluu.io/book/>
- Figures 10.1 to 10.4 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`).
- Terminal output: `src/ch09_gpt_model.py` .

Further reading (from the chapter)

- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language models are unsupervised multitask learners*. OpenAI technical report.

Dataset

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