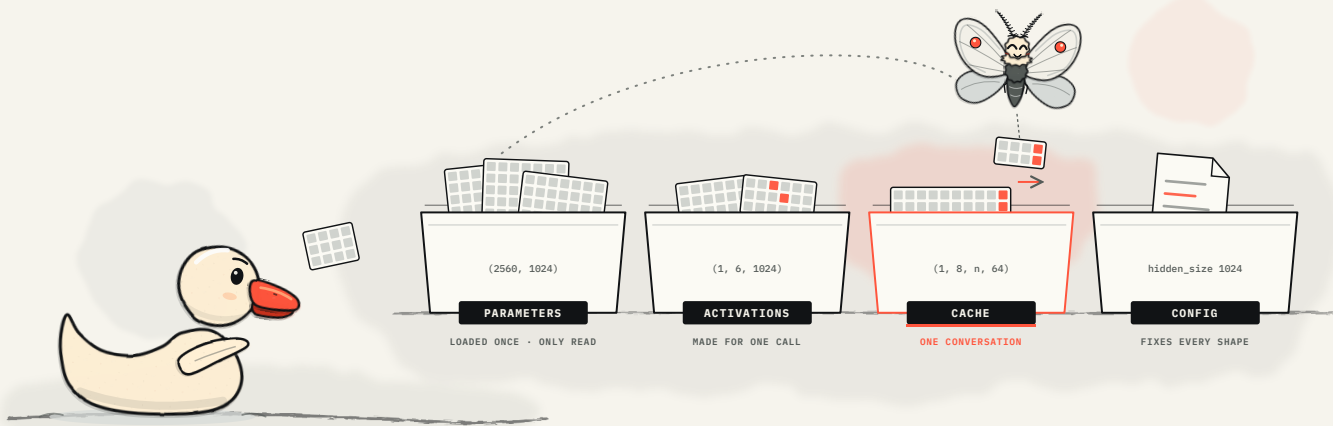




B1C02 · CHAPTER 2

Tensors, Weights, Layers, and State

IN THIS CHAPTER

What you will be able to do

- 1 Read an array in a model as a list of lists, say what each of its sizes counts (prompts, tokens, numbers per token, scores per vocabulary entry), and say which list of scores is the next-token guess.
- 2 Pick the slice you mean, every prompt's last token, and catch the wrong one from the size it returns, even when it prints believable tokens.
- 3 Read a learned array's name as a path and its size as rows of a given length, trace every size to a few numbers in the config file, explain why the saved numbers only load at those sizes, and count the learned numbers once when one array has two names.
- 4 Tell learned numbers from numbers made during a call: the made arrays grow with the input and are dropped, while the 132 learned arrays keep their size and are never written by running.
- 5 Explain why a reply keeps a cache and what it keeps: the work for earlier tokens never changes, attention blocks keep numbers for every token so far, conv blocks keep the last three positions, and a cache belongs to one conversation.
- 6 Sort any array a running model touches by what its sizes count, who made it and how long it lives, and name the chapter that opens each part.

WATCH · 1:12



Learned, or made while it runs

Chapter 1 ran a small open model, LiquidAI's LFM2.5-230M, on a laptop CPU. It ended on a question. Inside a running model, every number sits in an array. Which arrays did the model learn, and which exist only while it runs?

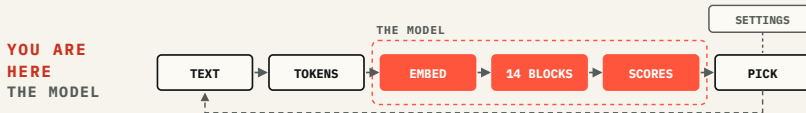
Some of each, plus one kind in between. A program you write already has all three. Some of its data is loaded at startup and never written. Some lives in a function's local variables, made during a call and gone after it. And some is state kept for one session. A running model has the same three lifetimes. Some arrays the model learned in training. It loads them from its file once and only reads them. Other arrays are made during one call to the model and dropped when the call is done. And while the model writes a reply, one kind of array is kept from each call to the next.

This chapter reads those arrays on the same model and sorts them. Reading comes first, because every array has sizes and each size counts something. Then come the learned arrays: their names, what their two sizes count, where those sizes come from, and how to count them. Then the arrays made for one call, and last the ones kept while a reply is written.

Sorting them pays off because mixing them up causes real bugs. We run three of them on this model, with real output. A slice of the model's output answers the wrong question and raises no error. A count of the learned numbers comes out 67 million too big. And a question about Spain gets the capital of France.



2.1 An array of arrays



Most arrays in a model are nested, and an array of arrays is something you already write. Here is a small one, built from plain Python lists:

■ LISTING 2-1 nested.py

```
# two prompts, three tokens each, four numbers per token
batch = [
    [[1, 0, 2, 5], [3, 1, 0, 0], [2, 2, 1, 4]], # prompt 0
    [[1, 0, 2, 5], [3, 1, 0, 0], [7, 0, 3, 1]], # prompt 1
]
print("prompts:           ", len(batch))
print("tokens per prompt:", len(batch[0]))
print("numbers per token:", len(batch[0][0]))
print("shape:", (len(batch), len(batch[0]), len(batch[0][0])))
print("prompt 1, token 2:", batch[1][2])
```

● OUTPUT OF LISTING 2-1

```
$ python3 nested.py
prompts:           2
tokens per prompt: 3
numbers per token: 4
shape: (2, 3, 4)
prompt 1, token 2: [7, 0, 3, 1]
```

The outer list holds two prompts. Each prompt holds three tokens, and each token is a list of four numbers. The two prompts differ only in their last token, the way "The capital of France is" and "The capital of Spain is" differ in one word. Asking `len` at each level reads off what that level counts: 2 prompts, 3 tokens per prompt, 4 numbers per token.



■ TECHNICAL STUFF

TERM axis

One level of a nested array. The first index you write picks along the first axis, the next index along the second.

TERM shape

The sizes of a nested array, from the outermost list in. Each size counts one thing, such as prompts, tokens, or numbers per token.

TERM tensor

What libraries call an array of numbers, often nested: a list of lists of lists. Every number in a running model sits in one.

TERM batch

Several independent prompts stacked into one array, so one call handles them all.

TERM position

Where a token sits in the sequence: 0 for the first token, 1 for the next.

TERM width

How many numbers describe one token inside the model: 1,024 here.

TERM embedding

The list of 1,024 numbers the model looks up for a token ID: one row of a big learned table.

`batch[1][2]` walks down two levels to prompt 1's token 2, which is one token's list of four numbers.

Each level of nesting has a name. An **axis** is one level of a nested array. The first index you write picks along the first axis, the next index along the second. The **shape** is the sizes of a nested array, from the outermost list in. Each size counts one thing, such as prompts, tokens, or numbers per token. Here the shape is (2, 3, 4). A **tensor** is what libraries call an array of numbers, often nested: a list of lists of lists. Every number in a running model sits in one. The word sounds more technical than the thing it names.

The three axes of a model's arrays have names of their own:

- ◆ The **batch** is several independent prompts stacked into one array, so one call handles them all. The batch axis says which prompt.
- ◆ The **position** is where a token sits in the sequence: 0 for the first token, 1 for the next. The position axis says which token.
- ◆ The **width** is how many numbers describe one token inside the model: 1,024 here. The width axis says which of a token's numbers.

Why nest them at all? Chapter 1 showed that the model turns each token into a list of numbers. An **embedding** is the list of 1,024 numbers the model looks up for a token ID: one row of a big learned table. A model handles many tokens in one call, and often several prompts. Lists of the same length stack into lists of lists, so one call can carry every token of every prompt. Stacking prompts does not mix them, as the next section shows.

Now the real model. The scripts in this chapter import the same `load.py` as chapter 1. It loads the pinned revision of the model with the Hugging Face Transformers library, which runs on PyTorch:



■ LISTING 2-2 load.py

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

REPO = "LiquidAI/LFM2.5-230M"
REV = "40cb2ad3b3044d5a41eee083a6103c8b523afa45"
tok = AutoTokenizer.from_pretrained(REPO, revision=REV)
model = AutoModelForCausalLM.from_pretrained(
    REPO, revision=REV, dtype=torch.bfloat16).eval()
```



■ TECHNICAL STUFF

The next script sends two prompts in one call. It prints the sizes of three arrays, then each prompt's top next token:

■ LISTING 2-3 shapes.py

```
import torch
from load import tok, model

texts = ["The capital of France is",
         "The capital of Spain is"]
ids = tok(texts, return_tensors="pt").input_ids
with torch.no_grad():
    out = model(ids, output_hidden_states=True)
emb = out.hidden_states[0] # the embedding output
print("token IDs ", tuple(ids.shape))
print("embeddings", tuple(emb.shape))
print("logits     ", tuple(out.logits.shape))
for b, text in enumerate(texts):
    top = int(out.logits[b, -1].argmax()) # prompt b, last position
    print(f"{text!r:28} -> {tok.decode([top])!r}")
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 2-3

```
$ python3 shapes.py
token IDs (2, 6)
embeddings (2, 6, 1024)
logits     (2, 6, 65536)
'The capital of France is' -> ' Paris'
'The capital of Spain is' -> ' Madrid'
```

The first two lines are the same levels as `nested.py`. The token IDs are 2 prompts of 6 tokens each. The six tokens include the start token, a special token with ID 1 that the tokenizer puts

before the text. The embeddings add a third level, the width. So that array is 2 prompts, 6 tokens each, and 1,024 numbers per token. The third line holds the scores, and the next section reads it.

A shape on its own does not say what an array is. (2, 3, 4) could count 2 images of 3 rows of 4 pixels, or 2 users with 3 orders of 4 items. The sizes mean something only once you say what each one counts. Before you index an array, say what each of its sizes counts.



■ QUICK CHECK

CHECK

In `nested.py`, what does `len(batch[1])` count, and what is it? It counts the tokens in prompt 1, and it is 3.

■ WATCH · 1:36



A score for every token, at every position

TERM scores

The raw numbers the output layer gives every possible next token, also called logits.

TERM vocabulary

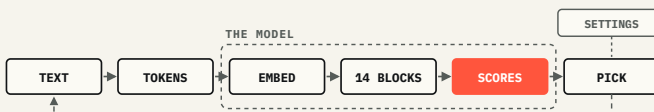
Every token the model can score: 65,536 of them for this model.



■ TECHNICAL STUFF

2.2 A score for every token, at every position

YOU ARE
HERE
THE SCORES



The model's last array holds its **scores**: the raw numbers the output layer gives every possible next token, also called logits. The **vocabulary** is every token the model can score: 65,536 of them for this model. Chapter 1 turned the last token's scores into probabilities and picked the next token from them. This section shows where that last list sits among the others. Here is a small version with a vocabulary of four words:

■ LISTING 2-4 slice.py

```

words = ["of", "is", "Paris", "Madrid"]
# scores[prompt][position][word]: one score per word
scores = [
    [[5, 1, 0, 0], [0, 6, 1, 0], [0, 0, 9, 2]], # France
    [[5, 1, 0, 0], [0, 6, 1, 0], [0, 0, 2, 9]], # Spain

```



slice.py CONTINUED

```

]
def best(row):
    return words[row.index(max(row))]

print("best at each position, Spain:",
      [best(row) for row in scores[1]])
right = [p[-1] for p in scores] # every prompt, last position
wrong = scores[-1]             # last prompt, every position
print("right:", len(right), "rows", [best(r) for r in right])
print("wrong:", len(wrong), "rows", [best(r) for r in wrong])

```

● OUTPUT OF LISTING 2-4

```

$ python3 slice.py
best at each position, Spain: ['of', 'is', 'Madrid']
right: 2 rows ['Paris', 'Madrid']
wrong: 3 rows ['of', 'is', 'Madrid']

```

The scores have the same three levels as `nested.py`, with a different inner level. Each prompt has three positions, and each position holds a list of four scores, one per word in the vocabulary. The best word at a position is the one with the highest score, which `best` finds as the index of the largest.

The first line of output is the best word at each position of the Spain prompt: `of`, `is`, `Madrid`. Read the Spain prompt's three positions as capital, Spain and is: after capital the best guess is `of`, after Spain it is `is`, and after `is` it is `Madrid`. Each position's list guesses the word that comes after that position. The next word of the reply is the guess at the last position, the word after the whole prompt.

The `right` line asks for that guess in every prompt at once. `[p[-1] for p in scores]` takes each prompt and keeps its last position. It returns 2 rows, one per prompt, and their best words are `Paris` and `Madrid`. A tensor library writes the same slice as `scores[:, -1]`. The colon means every entry on the first axis, and the `-1` picks the last entry on the second. The `wrong` line is the next section's subject.

Why does the model score every position when the reply needs only the last? The output layer treats every token's list the same way, so one call gives a guess after each position. The reply loop reads only the last one.

On the real model, the third line of `shapes.py`'s output is the scores. They are 2 prompts, 6 positions each, and at every position a score for each of the 65,536 tokens in the vocabulary. `out.logits[b, -1]` takes prompt `b`'s last position and keeps all 65,536 of its scores, and `argmax` picks the highest. Prompt `o` gets `Paris`, and prompt `1` gets `Madrid`.

Batching did not mix the two prompts. In a separate run we turned each prompt's last scores into probabilities, once in the batch and once alone. France gives `Paris` at 95.22 percent both ways, and Spain gives `Madrid` at 93.66 percent both ways. France's last scores in the batch were identical to its scores alone.

A natural first guess is that the scores hold one answer per prompt. They hold one list per position. Six tokens in gives six lists of scores out, for each prompt.



■ QUICK CHECK

CHECK

A third prompt is added to France and Spain, also six tokens long. Which size of the scores changes? Only the first, the batch, from 2 to 3. The positions stay at 6, and each position still holds 65,536 scores.

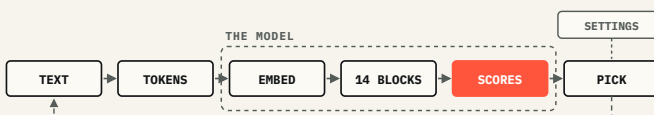
■ WATCH · 1:06



The wrong axis

2.3 The wrong axis

YOU ARE
HERE
THE SCORES



The right slice took one index. Index the other level by mistake and the code still runs. The `wrong` line of `slice.py` takes



`scores[-1]`, which indexes the outermost level. That picks the last prompt and keeps all of its positions. Its output is three real words, `of`, `is` and `Madrid`. They are the guesses after each position of the Spain prompt, and none of them answers what comes next in each prompt. The only sign is the size: 3 rows where 2 belong.

The real model makes the same slip look even more believable:

■ LISTING 2-5 wrong.py

```
import torch
from load import tok, model

texts = ["The capital of France is",
         "The capital of Spain is"]
ids = tok(texts, return_tensors="pt").input_ids
with torch.no_grad():
    logits = model(ids).logits
    # (batch, position, vocabulary)
right = logits[:, -1] # every prompt, last position
wrong = logits[-1] # last prompt, every position
print("right", tuple(right.shape))
print([tok.decode([int(i)]) for i in right.argmax(-1)])
print("wrong", tuple(wrong.shape))
print([tok.decode([int(i)]) for i in wrong.argmax(-1)])
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 2-5

```
$ python3 wrong.py
right (2, 65536)
[' Paris', ' Madrid']
wrong (6, 65536)
['<|startoftext|>', ' ', ' of', ' France', ' is', ' Madrid']
```

`logits[:, -1]` keeps every batch entry and takes the last position. That gives one next token per prompt, `Paris` and `Madrid`. `logits[-1]` indexes the first axis instead. It takes the last batch entry, the Spain prompt, and keeps all six positions. That gives six predictions, one after each position of the Spain prompt.

Both lines run without an error, and both print real tokens. The wrong one even looks plausible. Its fourth entry is the

model's guess after "The capital of", and the guess is `France`, at about 19 percent. A bug like this can slip past a quick look at the output.

The shape catches it. You wanted one row per prompt, so the first size should be 2. It is 6. When you index an array, write down the shape you expect, then compare it with the shape you got.



■ QUICK CHECK

CHECK

`scores[-1]` on 2 prompts of 6 tokens: how many lists of scores come back, and what do they score? Six lists, one guess after each position of the last prompt. None of them is the first prompt's answer.

■ WATCH • 1:37

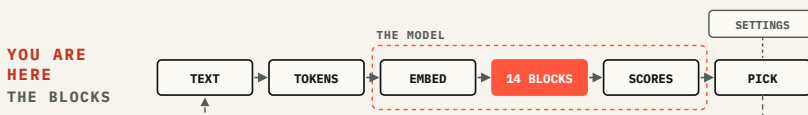


A weight's name is a path

TERM Parameters

The model's learned numbers, also called weights. Training sets them. Running the model only reads them. The next script lists them, then prints the names and sizes of everything outside the blocks, plus one conv block and one attention block:

2.4 Reading a weight's name



So far every array was made by running the model. The learned arrays are loaded from its file. Chapter 1 counted 229,693,184 parameters. **Parameters** are the model's learned numbers, also called weights. Training sets them. Running the model only reads them. They sit in named arrays. The next script lists them, then prints the names and sizes of everything outside the blocks, plus one conv block and one attention block:



■ LISTING 2-6 names.py

```

from load import model

params = list(model.named_parameters())
print(len(params), "tensors")
for name, p in params:
    block = name.split(".")[2] if ".layers." in name else ""
    if block in ("", "0", "2"):
        # one conv block, one attention block
        print(f"{name.removeprefix('model.')} :38} {tuple(p.shape)}")

```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 2-6

```

$ python3 names.py
132 tensors
embed_tokens.weight          (65536, 1024)
layers.0.conv.conv.weight    (1024, 1, 3)
layers.0.conv.in_proj.weight  (3072, 1024)
layers.0.conv.out_proj.weight (1024, 1024)
layers.0.feed_forward.w1.weight (2560, 1024)
layers.0.feed_forward.w3.weight (2560, 1024)
layers.0.feed_forward.w2.weight (1024, 2560)
layers.0.operator_norm.weight (1024,)
layers.0.ffn_norm.weight      (1024,)
layers.2.self_attn.q_proj.weight (1024, 1024)
layers.2.self_attn.k_proj.weight (512, 1024)
layers.2.self_attn.v_proj.weight (512, 1024)
layers.2.self_attn.out_proj.weight (1024, 1024)
layers.2.self_attn.q_layer_norm.weight (64,)
layers.2.self_attn.k_layer_norm.weight (64,)
layers.2.feed_forward.w1.weight (2560, 1024)
layers.2.feed_forward.w3.weight (2560, 1024)
layers.2.feed_forward.w2.weight (1024, 2560)
layers.2.operator_norm.weight (1024,)
layers.2.ffn_norm.weight      (1024,)
embedding_norm.weight         (1024,)

```

The model's 229,693,184 learned numbers sit in 132 arrays. A name reads like a path into nested objects, the way a JSON path does. Take `layers.2.self_attn.q_proj.weight`:

- ◆ `layers.2` is block 2, the third block. A **block** is one repeated stage that reworks every token's numbers. This model stacks 14 of them. The code calls them `layers`.

TERM block

One repeated stage that reworks every token's numbers. This model stacks 14 of them.



- ◆ `self_attn` is that block's attention part, its mixer.
- ◆ `q_proj` is one of attention's grids of learned numbers. Chapter 8 opens what each of them does.
- ◆ `weight` is the array of learned numbers in that part.

TERM mixer

The part of a block where tokens share information with each other.

TERM Attention

A mixer that lets each token pull information from every earlier token, weighted by content.

TERM Conv

A mixer that blends each token with the two tokens just before it (short for convolution).

TERM layer

A word with two meanings. In this model's code and config a layer is a block. People also use the word for one grid of learned numbers inside a block.

Every block has a mixer. A **mixer** is the part of a block where tokens share information with each other. This model has two kinds. **Attention** is a mixer that lets each token pull information from every earlier token, weighted by content. **Conv** is a mixer that blends each token with the two tokens just before it (short for convolution). Six of the 14 blocks use attention and the other eight use conv.

Not every array is a grid of rows. The embedding table is read as a lookup: each token ID selects one of its 65,536 rows. The norms are single lists of 1,024 or 64 numbers, and chapter 8 opens them. One name misleads. `embedding_norm.weight` is the final norm, which the model applies after the last block, not after the embeddings. A name is a strong hint, and the model's code is the check.

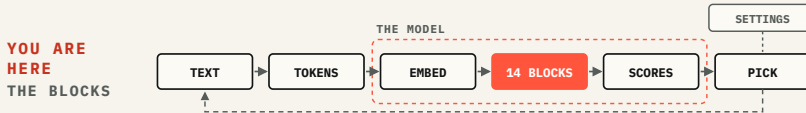
The word "layer" changes meaning from place to place. A **layer** is a word with two meanings. In this model's code and config a layer is a block. People also use the word for one grid of learned numbers inside a block. The config's `num_hidden_layers` is 14, and it counts blocks, each holding 8 or 11 arrays. When someone says "layer", read the name to see which one they mean.

Why does every array need a name? Every block of one kind repeats the same arrays at the same sizes. Block 0 is a conv block, whose mixer blends each token with the two tokens just before it, and every conv block holds the same 8 arrays: the conv filter (a learned row of 3 numbers for each of the 1,024 numbers, one per token it blends), a grid on each side of it, the three feed-forward grids and two norms. Block 2 is an attention block, and every attention block holds the same 11. The name is the only way to tell block 2's `q_proj` from block 4's. With the



embedding table and the final norm, that makes $1 + 8 \times 8 + 6 \times 11 + 1 = 132$ arrays.

2.5 Rows out, row length in



Most of the arrays in `names.py` have two sizes, which makes each one a grid. A grid's two sizes count different things. In code, a grid is a list of lists, and the script below reads the two lengths of a small one, two rows of three numbers:

■ LISTING 2-7 grid.py

```

w = [[2, 0, 1],           # row 0
     [1, 3, 0]]          # row 1
print("rows (numbers out):", len(w))
print("row length (numbers in):", len(w[0]))
print("takes a list of", len(w[0]), "numbers, gives", len(w))

```

● OUTPUT OF LISTING 2-7

```

$ python3 grid.py
rows (numbers out):      2
row length (numbers in): 3
takes a list of 3 numbers, gives 2

```

The outer length counts the rows, 2. The length of one row is 3. In a model, each row of such a grid makes one output number from the list it takes in, so a grid with 2 rows gives 2 numbers. Each row is as long as the list it takes in, so this grid takes 3. A learned grid like this has a name. A **weight matrix** is a grid of numbers the model learned in training. Each row makes one output number and is as long as the list it takes in. How a row

■ WATCH • 1:16



Rows out, row length in



■ TECHNICAL STUFF

TERM **weight matrix**

A grid of numbers the model learned in training. Each row makes one output number and is as long as the list it takes in.



turns a list into its one number is the whole of chapter 3. Here, read only the sizes: rows out, then row length in.

The real model's grids read the same way, because PyTorch stores each one with one row per output number. `w1` is 2,560 rows of 1,024. It takes a token's 1,024 numbers and gives 2,560. `w2` is 1,024 rows of 2,560, so it brings them back to 1,024. The two hold the same count of numbers, but the order of the sizes says which one widens a token's list and which narrows it back. That widening and narrowing is the **feed-forward**: the part of a block that reworks each token's numbers on its own, widening them and narrowing back.

TERM feed-forward

The part of a block that reworks each token's numbers on its own, widening them and narrowing back.



■ QUICK CHECK



■ QUICK CHECK

CHECK

`grid.py`'s grid has 2 rows of 3. How long is the list it takes, and how many numbers come out? It takes 3 numbers, one per place in a row, and gives 2, one per row.

CHECK

`layers.4.self_attn.k_proj.weight` is 512 rows of 1,024. Which block, which kind, and how many numbers go in and come out? Block 4, an attention block. It takes 1,024 numbers in and gives 512 out.

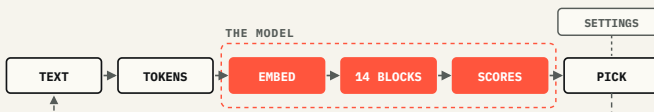
■ WATCH • 1:40



The config sets every size

2.6 The config sets every size

YOU ARE
HERE
THE MODEL



Who decided 2,560, or 1,024? In code, the answer is a constructor that allocates arrays from a dict of settings. The script below builds a small set of arrays from four settings. Then it edits one setting and tries to load the saved arrays into the new sizes:



■ LISTING 2-8 build.py

```

def grid(rows, cols):          # an empty grid of zeros
    return [[0] * cols for _ in range(rows)]

def build(config):              # every array, sized by config
    w, inner = config["width"], config["inner"]
    arrays = {"table": grid(config["vocab"], w)}
    for b in range(config["blocks"]):
        arrays[f"layers.{b}.w1"] = grid(inner, w)
        arrays[f"layers.{b}.w2"] = grid(w, inner)
    return arrays

def size(a):
    return f"{len(a)} rows of {len(a[0])}"

config = {"vocab": 5, "width": 3, "inner": 4, "blocks": 2}
saved = build(config)          # stands in for the saved file
for name, a in saved.items():
    print(f"{name:11} {size(a)}")

config["inner"] = 6            # edit one setting, then load
for name, a in build(config).items():
    if size(a) != size(saved[name]):
        print("cannot load", name)
        print("  saved:", size(saved[name]))
        print("  config wants:", size(a))
        break

```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 2-8

```

$ python3 build.py
table          5 rows of 3
layers.0.w1    4 rows of 3
layers.0.w2    3 rows of 4
layers.1.w1    4 rows of 3
layers.1.w2    3 rows of 4
cannot load layers.0.w1
  saved: 4 rows of 3
  config wants: 6 rows of 3

```

`build` makes empty grids of zeros, sized only by the settings. It makes a table of 5 rows of 3, and in each of 2 blocks a `w1` of 4 rows of 3 and a `w2` of 3 rows of 4. The first build stands in for the saved file. Then `inner` changes from 4 to 6, and the code builds



again. The first array whose size differs is `layers.0.w1`. It was saved as 4 rows of 3, and the edited settings want 6 rows of 3. The saved numbers have nowhere to go, so loading stops.

The settings hold no learned numbers. They only say how big each array is. The learned numbers fill the arrays, and they fit only the sizes they were saved at.

Real models are built this way for a reason. The same code runs this design at any size, and a short file picks the size. That is why a model ships with that file next to its weights. The **config** is the model's description file, `config.json`. Its numbers, like the width and the number of blocks, set the size of every learned array. It holds no learned numbers. Here is the real model's:

TERM **config**

The model's description file, `config.json`. Its numbers, like the width and the number of blocks, set the size of every learned array. It holds no learned numbers.



■ TECHNICAL STUFF

■ LISTING 2-9 `config.py`

```
from load import model

c = model.config
for k in ("vocab_size", "hidden_size", "intermediate_size",
         "num_hidden_layers", "num_attention_heads",
         "num_key_value_heads", "conv_L_cache"):
    print(f"{k:20} {getattr(c, k)}")
```

● OUTPUT OF LISTING 2-9

```
$ python3 config.py
vocab_size      65536
hidden_size     1024
intermediate_size 2560
num_hidden_layers 14
num_attention_heads 16
num_key_value_heads 8
conv_L_cache    3
```

TERM **settings file**

The file shipped with a model that sets default picking rules and the stop token (generation config).

The config is not the settings file from chapter 1. The **settings file** is the file shipped with a model that sets default picking rules and the stop token (generation config). This model ships both. `generation_config.json` holds the picking rules and the stop token, and `config.json` holds the sizes.



Match each config number to the sizes in `names.py`:

CONFIG VALUE	SIZES IT SETS
<code>vocab_size 65,536</code> and <code>hidden_size 1,024</code>	the embedding table, 65,536 rows of 1,024
<code>hidden_size 1,024</code>	the width: every block's input and output, the norms of 1,024, <code>out_proj</code> at 1,024 rows of 1,024
<code>intermediate_size 2,560</code>	<code>w1</code> and <code>w3</code> at 2,560 rows of 1,024, and <code>w2</code> at 1,024 rows of 2,560
<code>conv_L_cache 3</code>	the conv filter, 3 positions wide, one for each of the 1,024 numbers
<code>num_hidden_layers 14</code>	14 blocks, <code>layers.0</code> to <code>layers.13</code>

The conv filter is 3 positions wide because conv blends each token with the two before it, three tokens in all. The two attention settings set the sizes of attention's own grids, such as `k_proj`'s 512 rows, and chapter 8 opens them. Chapter 10 opens the conv block and its other grids.

The tempting mistake is to edit the config to change the model. The next script reads the real config, sets `intermediate_size` to 3,072, and loads the same weights under it:

■ LISTING 2-10 `wrongsize.py`

```
import textwrap
import torch
from transformers import AutoConfig, AutoModelForCausalLM
from load import REPO, REV

cfg = AutoConfig.from_pretrained(REPO, revision=REV)
print("intermediate_size in config:", cfg.intermediate_size)
cfg.intermediate_size = 3072      # edit one setting
try:
    AutoModelForCausalLM.from_pretrained(
        REPO, revision=REV, config=cfg, dtype=torch.bfloat16)
    print("loaded")
except RuntimeError as e:
    print(textwrap.fill("RuntimeError: " + str(e), 60))
```



■ TECHNICAL STUFF



● OUTPUT OF LISTING 2-10

```
$ python3 wrongsize.py
intermediate_size in config: 2560
RuntimeError: You set 'ignore_mismatched_sizes' to 'False',
thus raising an error. For details look at the above report!
```

Loading stopped with an error. With the library's logging turned on, its load report names the arrays that did not fit: `w1`, `w2` and `w3` in all 14 blocks. Each was saved with 2,560 where the edited config wanted 3,072. A bigger model needs new learned numbers, because the old ones only fit the sizes they were trained at.



■ QUICK CHECK

CHECK

Which config value would you change to make the feed-forward part wider, and which three arrays would change size?

`intermediate_size`. It sets the sizes of `w1`, `w3` and `w2`. The saved weights would no longer load, because they were trained at 2,560.

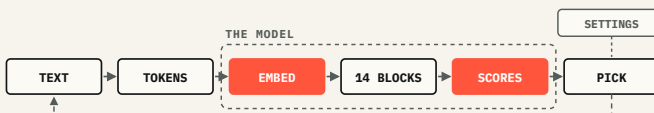
■ WATCH · 1:15



Count once

2.7 Counting once

YOU ARE
HERE
ONE TABLE,
TWO NAMES



Every learned array has a name and a size. Adding them all up should give the model's number of learned numbers. Done the obvious way, it does not. The reason is aliasing: two names for one list, which Python's `is` and `id()` can detect. Here it is in plain Python:



■ LISTING 2-11 alias.py

```

table = [[1, 0, 2], [0, 3, 1]]      # 6 numbers
saved = {"embed": table,           # the same list, two names
        "output": table}
by_name = sum(len(r) for t in saved.values() for r in t)
print("counted by name:", by_name)
print("same list:", saved["embed"] is saved["output"])
once = {id(t): t for t in saved.values()} # one entry per list
print("counted once: ", sum(len(r) for t in once.values()
                           for r in t))

```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 2-11

```

$ python3 alias.py
counted by name: 12
same list: True
counted once: 6

```

The dict holds one list under two names, `embed` and `output`. Summing over the names counts its 6 numbers twice, which gives 12. `saved["embed"] is saved["output"]` is `True`, so both names point to the same list. Keying a dict by `id()` keeps one entry per distinct list, and that count is 6. Count arrays, not names.

The real model has the same trap. Code that saves or loads a model works with its `state_dict`. The `state_dict` is the dict from name to array that saving code writes out. An array shared by two parts appears once under each name. The script below counts the parameters once, then sums the `state_dict`:

TERM state_dict

The dict from name to array that saving code writes out. An array shared by two parts appears once under each name.





■ TECHNICAL STUFF

■ LISTING 2-12 count.py

```
from load import model

params = sum(p.numel() for p in model.parameters())
saved = sum(t.numel() for t in model.state_dict().values())
print(f"parameters {params:>12,}")
print(f"state_dict {saved:>12,}")
sd = model.state_dict()
print("lm_head.weight listed:", "lm_head.weight" in sd)
print("same storage:", sd["lm_head.weight"].data_ptr()
      == sd["model.embed_tokens.weight"].data_ptr())
```

● OUTPUT OF LISTING 2-12

```
$ python3 count.py
parameters    229,693,184
state_dict    296,802,048
lm_head.weight listed: True
same storage: True
```



The `state_dict` holds 67,108,864 more numbers, exactly one table of 65,536 rows of 1,024. This model's output layer reuses the embedding table instead of holding its own, and chapter 7 explains why. The `state_dict` lists that one array under two names, `model.embed_tokens.weight` and `lm_head.weight`, so summing it counts the table twice. `model.parameters()` gives each shared array once, which is how it reaches 229,693,184.

Equal values would not prove the two names are one array, because two separate copies could hold the same numbers. The last line checks the address in memory where each array's numbers start. Both names point to the same address, so they are the same array, as is showed for the small dict.



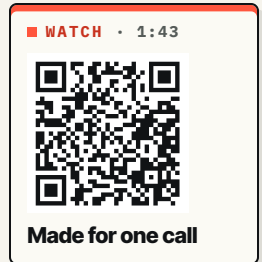
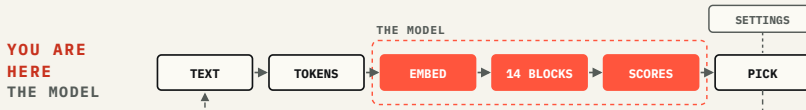
■ QUICK CHECK

CHECK

Why does summing the saved listing give 296,802,048? The output layer and the embedding table are one array under two names. The sum counts its 67,108,864 numbers twice.



2.8 Made for one call



The learned arrays are what the model holds. Everything else is made while it runs. In code, the difference is a function's local variables against the data it only reads. The function below reads a table of 15 learned numbers. For each word it makes two lists of three: the word's row of the table, and that row added to the previous word's row. It runs on 5 words, then on 50:

■ LISTING 2-13 made.py

```

table = {"the": [1, 0, 2], "capital": [0, 3, 1],
        "of": [2, 2, 0], "France": [1, 4, 1], "is": [0, 1, 1]}
learned = sum(len(row) for row in table.values())

def call(words):
    # reads the table, makes lists
    made, prev = [], [0, 0, 0]
    # made: numbers made this call
    for w in words:
        row = list(table[w])
        # a copied row
        mixed = [a + b for a, b in zip(row, prev)]
        made += [row, mixed]
        prev = row
    return made

before = {w: list(r) for w, r in table.items()}
for text in ("the capital of France is",
            "the capital of France is " * 10):
    n = sum(len(x) for x in call(text.split()))
    print(f"{len(text.split()):>2} words made {n:>3} numbers"
          f" learned {learned}")
print("table unchanged:", table == before)
  
```



■ TECHNICAL STUFF



● OUTPUT OF LISTING 2-13

```
$ python3 made.py
5 words made 30 numbers learned 15
50 words made 300 numbers learned 15
table unchanged: True
```

TERM **activation**

An array the model computes during one call, like a block's output. Made fresh each call, then dropped.

The function `made` 30 numbers for 5 words and 300 for 50. What it makes grows with the input. The table stayed at 15 numbers, and the last line confirms that no call changed it. The function only reads it.

The model's arrays split the same way. An **activation** is an array the model computes during one call, like a block's output. Made fresh each call, then dropped. The embeddings, every block's output, the 2,560 numbers per token inside each feed-forward part and the scores at the end are all activations.

The next script counts them. It registers a callback on every layer that holds weights. PyTorch calls the callback each time that layer runs, and here the callback adds up how many numbers the layer hands on. PyTorch names this kind of callback a forward hook. The script runs a 6-token prompt and a 352-token prompt:



■ TECHNICAL STUFF

■ LISTING 2-14 activations.py

```
import torch
from load import tok, model

made = [] # numbers each layer hands on
def hook(module, inputs, out):
    made.append(out.numel())
for m in model.modules(): # every layer that holds weights
    if list(m.parameters(recurse=False)):
        m.register_forward_hook(hook)

params = sum(p.numel() for p in model.parameters())
for text in ("The capital of France is",
             "The capital of France is Paris. " * 50):
    ids = tok(text, return_tensors="pt").input_ids
    made.clear()
    with torch.no_grad():
        model(ids)
```

CONTINUES OVERLEAF →



activations.py CONTINUED

```
print(f"{ids.shape[1]:>4} IDs  activations {sum(made):>12,}"
      f"  parameters {params:,}")
```

● OUTPUT OF LISTING 2-14

```
$ python3 activations.py
   6 IDs  activations    1,456,128  parameters 229,693,184
 352 IDs  activations   85,426,176  parameters 229,693,184
```

The parameters did not change with the prompt. They are 229,693,184 both times. The activations grew with it, from about 1.5 million numbers to about 85 million, which is 242,688 numbers per position both times. Every activation has a position axis, so a longer prompt makes each one longer, and a bigger batch multiplies them again. The parameters have no position axis, so one set serves every prompt. The output layer alone hands on 65,536 numbers per position, more than a quarter of the total, because it scores every vocabulary entry.

These are numbers handed from one layer to the next over the call, not memory in use at one moment. Most activations are released as soon as the next layer has read them. The count also leaves out arrays made inside a single step, such as the grid each attention block builds to compare every token with every earlier one: 576 numbers at 6 tokens, almost 2 million at 352. Chapter 13 measures what a model needs in memory.

Running the model reads the parameters and never writes them. Chapter 1 showed that for one call. The next script makes a stronger test of it. It copies all 132 arrays, generates three slogans with the shipped picking rules, and compares:

■ LISTING 2-15 readonly.py

```
import torch
from load import tok, model

before = {n: p.clone() for n, p in model.named_parameters()}
Q = "Write a one-line slogan for a coffee shop."
msgs = [{"role": "user", "content": Q}]
```

CONTINUES OVERLEAF →



■ TECHNICAL STUFF



readonly.py CONTINUED

```

ids = tok.apply_chat_template(
    msgs, add_generation_prompt=True,
    return_tensors="pt", tokenize=True)["input_ids"]
for seed in range(3):
    torch.manual_seed(seed)
    model.generate(ids, max_new_tokens=40)
same = sum(torch.equal(before[n], p)
            for n, p in model.named_parameters())
print(same, "of", len(before), "tensors unchanged")

```

● OUTPUT OF LISTING 2-15

```

$ python3 readonly.py
132 of 132 tensors unchanged

```



Every number is exactly as it was. A model does not learn from the prompts you send it. Training changes parameters, and chapter 6 shows how. Running only makes activations, and, while a reply is written, a cache.



■ WATCH OUT

CHECK

Does a 352-token prompt use more parameters than a 6-token prompt? No. Both calls read the same 229,693,184 parameters. The 352-token prompt makes about 59 times more activations.

■ WATCH · 1:30

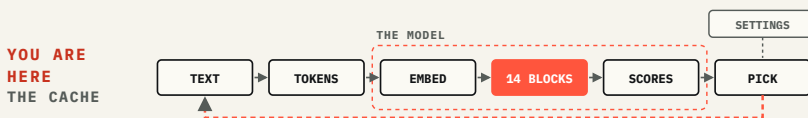


Redo or remember

TERM loop

Call the model, pick one token, append it to the input, and call again, until a stop token appears.

2.9 Redo or remember



A reply takes many calls. Chapter 1's **loop** was this: call the model, pick one token, append it to the input, and call again, until a stop token appears. Each call sent the whole text again,



so the model redid the work for every earlier token at every step. Can it keep that work instead?

In code, keeping a step's result so a later step does not recompute it is called memoization. Here are two versions of a small loop. Each token's work is to add it to a running total, and the next token is the total's last digit. `redo` resends every token at every step. `remember` keeps each token's total and adds only the new token:

■ LISTING 2-16 redo.py

```
prompt = [3, 1, 4]
def step(prev, t):           # one token's work:
    return prev + t          # its running total

def redo(n):                 # resend everything
    seq, work = list(prompt), 0
    for _ in range(n):
        total = 0
        for t in seq:
            total = step(total, t); work += 1
        seq.append(total % 10)
    return seq, work

def remember(n):             # keep each result: a cache
    seq, cache, work, new = list(prompt), [0], 0, list(prompt)
    for _ in range(n):
        for t in new:
            cache.append(step(cache[-1], t)); work += 1
        new = [cache[-1] % 10]
        seq += new
    return seq, work

for name, f in (("redo", redo), ("remember", remember)):
    seq, work = f(5)
    print(f"{name:8} tokens {seq} work {work}")
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 2-16

```
$ python3 redo.py
redo      tokens [3, 1, 4, 8, 6, 2, 4, 8] work 25
remember tokens [3, 1, 4, 8, 6, 2, 4, 8] work 7
```



Both write the same tokens. Redoing costs 25 units of work, and remembering costs 7. Keeping the earlier work changed the cost, not the answer.

TERM State

Data the model carries from one call to the next, like the cache.

TERM cache

What the model keeps between steps of one reply, so earlier tokens are not recomputed. It belongs to one conversation.

What a program keeps from one call to the next has a name. **State** is data the model carries from one call to the next, like the cache. The **cache** is what the model keeps between steps of one reply, so earlier tokens are not recomputed. It belongs to one conversation.

Why can a model keep its earlier work? In this model, each token's numbers depend only on that token and the tokens before it. Attention lets a token look back at earlier tokens, never ahead. The conv part reads a token and the two before it. So appending a token never changes the work already done for the earlier ones, and that work can be kept instead of redone. This is a property of this model's design. Chapter 21 meets a model that does not write left to right.

On the real model, the next script runs 64 steps of a reply from chapter 1's 21-token chat prompt, always taking the top token. It runs them both ways, counts the tokens sent through the model, and checks that the two replies match:



■ TECHNICAL STUFF

■ LISTING 2-17 saved.py

```
import torch
from load import tok, model

Q = "What is the capital of France? Answer in one sentence."
msgs = [{"role": "user", "content": Q}]
ids = tok.apply_chat_template(msgs, add_generation_prompt=True,
                             return_tensors="pt")["input_ids"]

def run(steps, keep):
    cache, new, sent, toks = None, ids, 0, []
    for _ in range(steps):
        with torch.no_grad():
            out = model(new, past_key_values=cache,
                       use_cache=keep)
        sent += new.shape[1]
        nxt = out.logits[0, -1].argmax().view(1, 1)
        toks.append(int(nxt))
    if keep:
        # remember: cache + one token
```

CONTINUES OVERLEAF →



saved.py **CONTINUED**

```

        cache, new = out.past_key_values, nxt
    else:
        # redo: resend everything
        new = torch.cat([new, nxt], 1)
    return toks, sent

redo, remember = run(64, False), run(64, True)
print("64 steps, redo:      tokens sent", redo[1])
print("64 steps, remember: tokens sent", remember[1])
print("same 64 tokens:", redo[0] == remember[0])

```

● OUTPUT OF LISTING 2-17

```

$ python3 saved.py
64 steps, redo:      tokens sent 3360
64 steps, remember: tokens sent 84
same 64 tokens: True

```

Resending everything sent 3,360 tokens through the model: 21 on the first step, 22 on the next, and so on up to 84. Keeping the cache sent 84: the 21-token prompt once, then one token per step. Both wrote the same 64 tokens. This counts tokens, not time. How much time a cache saves depends on the runtime and the hardware. Chapter 9 covers the reply loop with its cache, and chapter 13 covers the cost.

CHECK

Can a correct cache change the reply? No. Here the cached and the resending loops wrote the same 64 tokens, because the kept work is exactly the work the resending loop redid.

**■ WATCH OUT**

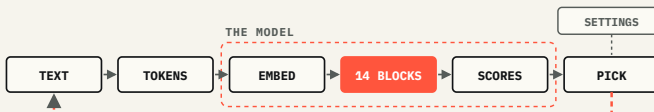
■ WATCH • 1:54



What the cache
keeps

2.10 What the cache keeps

YOU ARE
HERE
THE CACHE



What a block has to keep depends on how far back it looks. In code, that is the difference between a list that keeps growing and a fixed-size window. The script below runs two rules one token at a time from the prompt `[3, 1, 4]`. Rule A reads every token so far and counts how many are larger than the newest. Rule B reads only the last three and takes the last digit of their sum, which becomes the next token:



■ TECHNICAL STUFF

■ LISTING 2-18 lookback.py

```

everything = []                # every token so far
window = []                   # only the last three
new = [3, 1, 4]               # the prompt
for step in range(5):
    for t in new:
        everything.append(t)
        window = (window + [t])[-3:]
    # rule A reads every token: how many beat the newest?
    a = sum(1 for t in everything if t > everything[-1])
    # rule B reads the last three: their sum, last digit
    b = sum(window) % 10
    print(f"step {step} sent {len(new)} | A kept "
          f"{len(everything)} says {a} | B kept {len(window)}"
          f" says {b}")
    new = [b]
  
```



● OUTPUT OF LISTING 2-18

```
$ python3 lookback.py
step 0 sent 3 | A kept 3 says 0 | B kept 3 says 8
step 1 sent 1 | A kept 4 says 0 | B kept 3 says 3
step 2 sent 1 | A kept 5 says 2 | B kept 3 says 5
step 3 sent 1 | A kept 6 says 1 | B kept 3 says 6
step 4 sent 1 | A kept 7 says 1 | B kept 3 says 4
```

Rule A's kept list grows by one every step, from 3 to 7. Rule B's window stays at 3. After the first call, each call sends one token. What a rule must keep depends on how far back it looks.

The model has both kinds of rule. **Attention** is a mixer that lets each token pull information from every earlier token, weighted by content. So an attention block keeps numbers for every token so far. **Keys and values** are the numbers an attention block keeps for every token so far, so each new token can look back at them. **Conv** is a mixer that blends each token with the two tokens just before it (short for convolution). So a conv block keeps a fixed window. Its **conv state** is what a conv block keeps between calls: the numbers it blends, for the last three positions.

The next script runs the cached loop on chapter 1's question. At each step it prints the tokens it sent, the token it picked, and what one attention block and one conv block keep:

■ LISTING 2-19 cache.py

```
import torch
from load import tok, model

Q = "What is the capital of France? Answer in one sentence."
msgs = [{"role": "user", "content": Q}]
ids = tok.apply_chat_template(
    msgs, add_generation_prompt=True,
    return_tensors="pt", tokenize=True)["input_ids"]
cache, new = None, ids # first call: the prompt
for step in range(40):
    with torch.no_grad():
        out = model(new, past_key_values=cache, use_cache=True)
        cache = out.past_key_values # kept for the next call
        nxt = int(out.logits[0, -1].argmax())
```

CONTINUES OVERLEAF →

TERM Keys and values

The numbers an attention block keeps for every token so far, so each new token can look back at them.

TERM conv state

What a conv block keeps between calls: the numbers it blends, for the last three positions.



■ TECHNICAL STUFF

cache.py CONTINUED

```

keys = cache.layers[2].keys          # attention block 2
conv = cache.layers[0].conv_states[0] # conv block 0
print(f"{step} in {new.shape[1]:>2} {tok.decode([nxt])!r:12}"
      f" keys {tuple(keys.shape)} conv {tuple(conv.shape)}")
if nxt == 7:
    break
new = torch.tensor([[nxt]])          # next call: one token

```

● OUTPUT OF LISTING 2-19

```

$ python3 cache.py
0 in 21 'The'          keys (1, 8, 21, 64) conv (1, 1024, 3)
1 in 1 ' capital'     keys (1, 8, 22, 64) conv (1, 1024, 3)
2 in 1 ' of'          keys (1, 8, 23, 64) conv (1, 1024, 3)
3 in 1 ' France'      keys (1, 8, 24, 64) conv (1, 1024, 3)
4 in 1 ' is'          keys (1, 8, 25, 64) conv (1, 1024, 3)
5 in 1 ' Paris'       keys (1, 8, 26, 64) conv (1, 1024, 3)
6 in 1 ' .'          keys (1, 8, 27, 64) conv (1, 1024, 3)
7 in 1 '<|im_end|>'   keys (1, 8, 28, 64) conv (1, 1024, 3)

```

The reply is the same eight tokens as chapter 1's loop: The capital of France is Paris. and the end-of-turn token. The difference is the `in` column. The first call sends the 21-token prompt, and every later call sends one token.

Block 2 is an attention block. Its keys are 1 prompt, 8 groups, 21 tokens so far, and 64 numbers per group. So for every token so far it keeps 512 numbers of keys, and as many of values. The token count grows by one each step, from 21 to 28. Block 0 is a conv block. Its conv state is 1 prompt, 1,024 numbers, and 3 positions. It never grows. Each step drops the oldest position and adds the newest.

These two arrays are a reason to say what each size counts instead of assuming where it sits. In the keys, the token level comes before the 64 numbers of each group. In the conv state, the 1,024 numbers come first and the 3 positions last.

Across the whole model, each of the six attention blocks keeps $2 \times 8 \times 64 = 1,024$ numbers per token, 6,144 per token in all. The eight conv blocks keep 24,576 numbers in total, however long the reply runs. We calculated that at 10,000 tokens the



cache would hold about 61 million numbers, about a quarter as many as the parameters.

CHECK

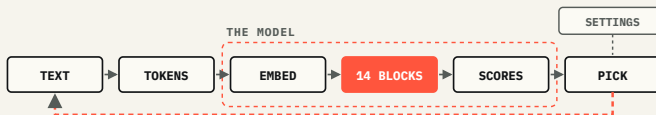
From the 21-token prompt, after the prompt call and ten one-token calls, how many tokens are in block 2's keys, and how many positions in block 0's conv state? 31 tokens in the keys, and still 3 positions in the conv state.



■ QUICK CHECK

2.11 A cache belongs to one conversation

YOU ARE
HERE
THE CACHE



■ WATCH • 1:04



The wrong cache

The cache stands in for every earlier token. The model never sees those tokens again, only what the cache kept from them. So whatever is in the cache, the model takes as the conversation so far. In code, the danger is a cache stored under the wrong key. The server below keeps earlier words between requests. It keys them first by the prompt's word count, then by conversation:

■ LISTING 2-20 session.py

```
capitals = {"France": "Paris", "Spain": "Madrid"}

def model(words):
    # reads every word it is given
    for w in words:
        if w in capitals:
            return capitals[w]

caches = {}
# kept between requests
def serve(key, words):
    earlier = caches.setdefault(key, words[:-1])
    return model(earlier + words[-1:]) # cache + last word

fr = "The capital of France is".split()
```

CONTINUES OVERLEAF →



■ TECHNICAL STUFF



session.py **CONTINUED**

```

es = "The capital of Spain is".split()
print("keyed by length:      ",
      serve(len(fr), fr), serve(len(es), es))
caches.clear()
print("keyed by conversation:",
      serve("chat 1", fr), serve("chat 2", es))

```

● **OUTPUT** OF LISTING 2-20

```

$ python3 session.py
keyed by length:      Paris Paris
keyed by conversation: Paris Madrid

```

Both prompts are 5 words long. Keyed by length, the Spain request finds France's stored words and answers **Paris**. Keyed by conversation, each request gets its own stored words, and the answers are **Paris** and **Madrid**. The function believes whatever words it is handed.

The real model does the same:

■ **TECHNICAL STUFF**■ **LISTING 2-21** stale.py

```

import torch
from load import tok, model

def ids(text):
    return tok(text, return_tensors="pt").input_ids
def top(out):
    return tok.decode([int(out.logits[0, -1].argmax())])

fr = ids("The capital of France is")
es = ids("The capital of Spain is")
with torch.no_grad():
    print("Spain, whole prompt: ", repr(top(model(es))))
    cache = model(fr[:, :-1], use_cache=True).past_key_values
    out = model(es[:, -1:], past_key_values=cache)
    print("Spain, France's cache:", repr(top(out)))

```



● OUTPUT OF LISTING 2-21

```
$ python3 stale.py
Spain, whole prompt: ' Madrid'
Spain, France's cache: ' Paris'
```

WATCH IT GO WRONG

The two prompts are both six tokens long, counting the start token, and they differ only at position 4, **France** against **Spain**. The script builds a cache from the first five tokens of the France prompt. Then it sends the last token of the Spain prompt, **is**, with that cache. The model answers **Paris**, with exactly the probabilities it gives the France prompt, 95.22 percent for **Paris**. As far as the model can tell, it was asked about France.

A runtime that reuses a cache because the earlier tokens have the same count, without checking that they are the same tokens, makes this bug. So does one that hands one conversation's cache to another. The parameters are fine, and the model did exactly its job.



A cache is part of the input. Start each conversation with an empty cache, and reuse one only for the exact tokens it was built from.

CHECK

What goes wrong if two conversations share a cache? The model answers the other conversation's question, as it answered Paris for Spain here. Two users can share the parameters, which are only read. Each needs a cache of their own.



■ QUICK CHECK



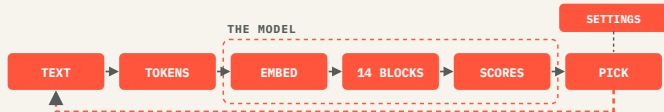
■ WATCH • 1:16



Sort any array

2.12 Sorting any array

YOU ARE
HERE
THE WHOLE
MACHINE



Three questions place any array a running model touches:

1. **What do its sizes count?** Say it before you index the array.
2. **Who made it?** Training made the parameters. One call made the activations. The loop made the cache from activations. The publisher wrote the config.
3. **How long does it live?** As long as the model is loaded, for one call, or for one conversation.



■ REMEMBER

parameter	EXAMPLE layers.0.feed_forward.w1.weight MADE BY training, then loaded from the file	LIVES as long as the model is loaded	SIZES 2,560 rows of 1,024 CHANGES WHILE RUNNING? no
activation	EXAMPLE block 5's output for the France prompt MADE BY one call	LIVES until the next layer has used it, the call at most	SIZES 1 prompt, 6 tokens, 1,024 numbers each CHANGES WHILE RUNNING? made fresh each call
cache	EXAMPLE block 2's keys MADE BY the loop, from activations	LIVES one conversation	SIZES 1 prompt, 8 groups, the tokens so far, 64 numbers per group CHANGES WHILE RUNNING? grows one token per step



cache	EXAMPLE block 0's conv state		SIZES 1 prompt, 1,024 numbers, 3 positions
	MADE BY the loop, from activations	LIVES one conversation	CHANGES WHILE RUNNING? overwritten each step, never grows
config	EXAMPLE hidden_size		SIZES a number, not an array
	MADE BY the publisher	LIVES ships with the model	CHANGES WHILE RUNNING? no

Two kinds of array sit outside the table. The token IDs are the input the tokenizer hands the model. And when the model loads, it computes two small lists of 32 numbers from the config, which it uses to mark positions. They are not parameters, and training did not set them.

The questions also find this chapter's bugs. The wrong slice failed the first question, because its first size counted positions where it should have counted prompts. The stale cache failed the third, because it was used outside the conversation it lives for. The double count needs one more check for learned arrays: are two names the same array?

CHECK

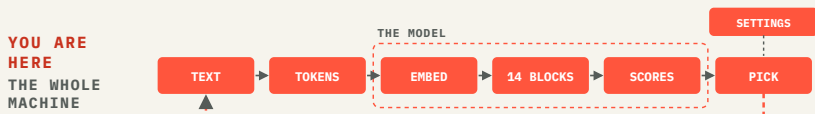
A runtime holds an array of shape (1, 8, 412, 64) between calls. Which kind is it, and what does 412 count? It is a cache, the keys or values of an attention block. 412 counts the tokens the conversation has reached so far.



■ QUICK CHECK

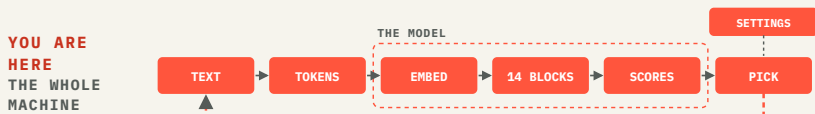


2.13 The map



PART	OPENS IN
What a weight matrix computes, row by row	B1C03
How each number is stored in two bytes	B1C04
How training sets the parameters	B1C06
Why the output layer shares the embedding table	B1C07
Keys, values, groups of 64, and the norms	B1C08
The cache in the reply loop, reset and sharing	B1C09
The conv block and its window of three	B1C10
Array names in the weights file	B1C11
Memory and time for parameters, activations and the cache	B1C13

2.14 Put it together



Every number in a running model sits in an array, a list of lists. Each level of nesting is an axis, and each size counts one thing, such as prompts, tokens or numbers per token. The scores hold one list per position, and only the last position's list is the next-token guess, so the slice you want is every prompt's last position. The wrong slice runs and prints real tokens, and only its size gives it away. The model's learned numbers sit in 132 named arrays. A name is a path to a block



and a part, and a weight matrix's size is its rows (numbers out) by its row length (numbers in). A few numbers in the config file set every size, and the learned numbers only load at the sizes they were trained at. One table is saved under two names, so count arrays, not names. Every call makes activations that grow with the input and are dropped, while the parameters are only read. In this model, adding a token never changes the work done for earlier tokens, so a reply keeps that work in a cache and sends one new token per call. Attention blocks keep keys and values for every token so far, and conv blocks keep the last three positions. The cache stands in for the conversation, so it belongs to exactly one.

The next chapter opens the weight matrices. A token's 1,024 numbers go into `w1`, a grid the model learned, and 2,560 numbers come out. How does each row turn a token's 1,024 numbers into its one number?

■ Further reading

- ◆ PyTorch documentation: `torch.Tensor`, `torch.nn.Module` (`named_parameters`, `state_dict`, `register_forward_hook`), and `torch.nn.Linear`.
- ◆ Hugging Face Transformers documentation: "Caching" and the LFM2 model page, and the `modeling_lfm2.py` source for the attention and conv caches.



Cheat Sheet



THE MACHINE • WHERE EACH PART IS TAUGHT

2.1 An array of arrays



2.2 A score for every token, at every position



2.3 The wrong axis



2.4 Reading a weight's name



2.5 Rows out, row length in



2.6 The config sets every size



2.7 Counting once



2.8 Made for one call



2.9 Redo or remember



2.10 What the cache keeps



2.11 A cache belongs to one conversation



TERMS • 27 IN THIS CHAPTER

parameters The model's learned numbers, also called weights. Training sets them. Running the model only reads them.

axis One level of a nested array. The first index you write picks along the first axis, the next index along the second.

shape The sizes of a nested array, from the outermost list in. Each size counts one thing, such as prompts, tokens, or numbers per token.

tensor What libraries call an array of numbers, often nested: a list of lists of lists. Every number in a running model sits in one.

batch Several independent prompts stacked into one array, so one call handles them all.

position Where a token sits in the sequence: 0 for the first token, 1 for the next.

width How many numbers describe one token inside the model: 1,024 here.

embedding The list of 1,024 numbers the model looks up for a token ID: one row of a big learned table.

scores The raw numbers the output layer gives every possible next token, also called logits.

vocabulary Every token the model can score: 65,536 of them for this model.

block One repeated stage that reworks every token's numbers. This model stacks 14 of them.

mixer The part of a block where tokens share information with each other.

attention A mixer that lets each token pull information from every earlier token, weighted by content.

conv A mixer that blends each token with the two tokens just before it (short for convolution).

layer A word with two meanings. In this model's code and config a layer is a block. People also use the word for one grid of learned numbers inside a block.

weight matrix A grid of numbers the model learned in training. Each row makes one output number and is as long as the list it takes in.

feed-forward The part of a block that reworks each token's numbers on its own, widening them and narrowing back.

config The model's description file, config.json. Its numbers, like the width and the number of blocks, set the size of every learned array. It holds no learned numbers.

settings file The file shipped with a model that sets default picking rules and the stop token (generation config).

state_dict The dict from name to array that saving code writes out. An array shared by two parts appears once under each name.

activation An array the model computes during one call, like a block's output. Made fresh each call, then dropped.

loop Call the model, pick one token, append it to the input, and call again, until a stop token appears.

state Data the model carries from one call to the next, like the cache.

cache What the model keeps between steps of one reply, so earlier tokens are not recomputed. It belongs to one conversation.

keys and values The numbers an attention block keeps for every token so far, so each new token can look back at them.

conv state What a conv block keeps between calls: the numbers it blends, for the last three positions.

start token A special token, ID 1, placed before your text to mark the start of a sequence.

