

THE LOOP

```
for i in range(len(q)):
    total += q[i] * b[i]
```



TOP 5 OF 65,536

Paris	21.00
Re	15.88
Strasbourg	15.69
Bordeaux	15.31
Lyon	15.12

TWO LISTS

MULTIPLY PAIR BY PAIR

MATCH SCORE

ADD THE PRODUCTS

THE SAME MOVE

ONE SCORE PER TOKEN

B1C03 • CHAPTER 3

Linear Algebra Inside a Model

IN THIS CHAPTER

What you will be able to do

- 1 Say why a model holds a word as a list of numbers, not as text: text matching follows spelling, while the model's lists put related words close together, and the blocks then rework a word's list using the words around it.
- 2 Write a dot product as a loop, and say what its answer means: a score for how well two lists match.
- 3 Show that the model's score for `Paris` is one dot product of two lists of 1,024 numbers.
- 4 Explain a matrix as an array of arrays and matrix multiplication as a nested loop, and show that the model's next word is the best match among the 65,536 rows of its output table.
- 5 Explain how a matrix turns one list into another, show that all 83 of the model's weight matrices do exactly this, and show that the lists in a batch never mix inside one.
- 6 Check that lengths line up, explain which side of a grid a list meets and why the code therefore reads a stored matrix on its side, and catch the square-matrix mistake that a length check misses.
- 7 Count the work: one multiply-add per weight, per token.

WATCH • 1:22



One piece of arithmetic, repeated



Chapter 2 ended on a question. What actually happens when a token's 1,024 numbers meet a weight matrix, one of the grids of learned numbers inside the model? The answer is one piece of arithmetic, repeated all through the model. The model takes two lists of numbers, multiplies them pair by pair, and adds up the products. That one number scores how well the two lists match. Chapter 1's answer, `Paris` at 95 percent after "The capital of France is", is the best of 65,536 of those scores.

This chapter builds that multiply-and-add from nothing. It starts with why the model holds a word as a list of numbers at all. Each idea starts as code you already write, such as a loop over two arrays or an array of arrays. It runs first on a few small numbers you can see whole, and we say what the result means. Then the same idea runs on the real model, LiquidAI's LFM2.5-230M on a laptop CPU, as proof. Math has names for these ideas: vector, dot product, matrix, matrix multiplication, projection and transpose. Each name arrives after the thing it names.

Knowing that one piece of arithmetic pays off in three ways. You can say what any weight matrix computes and how long its output is before you run it. You can count how much arithmetic the model does for each token. And you can catch a real bug that no error message reports. At the end of the chapter, one square matrix read the wrong way round runs without an error, and the model stops saying Paris.

■ WATCH • 1:41



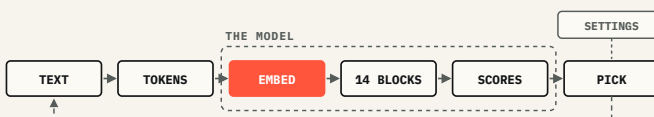
A word is a list of numbers

TERM token

A small piece of text, often a word or part of one, that the model reads and writes one at a time.

3.1 A word is a list of numbers

YOU ARE
HERE
THE
EMBEDDING
TABLE



Start with what the model holds when it holds a word. It does not hold text. It holds an array of numbers. A **token** is a small



piece of text, often a word or part of one, that the model reads and writes one at a time.

Here is a small array, three numbers long:

```
[2, 0, 1]
```

A **vector** is a list of numbers. One token's 1,024 numbers are a vector. In code, a plain array. That is the whole definition, and nothing in this chapter needs more.

Chapter 1 showed how the model gets from text to these lists. The tokenizer turns each token into a token ID, a whole number. Each ID then picks one row of a big table the model learned in training. An **embedding** is the list of 1,024 numbers the model looks up for a token ID: one row of a big learned table.

The scripts in this chapter import the same `load.py` as chapters 1 and 2. It loads the pinned revision of the model with the Hugging Face Transformers library, on top of PyTorch, the library that does the arithmetic:

■ LISTING 3-1 `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()
```

The script below looks up `Paris` (with its leading space, as the tokenizer writes it). It reads that token's row of the embedding table, turns it into a plain Python list, and prints how long it is and a few of its numbers:

TERM **vector**

A list of numbers. One token's 1,024 numbers are a vector. In code, a plain array.

TERM **embedding**

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



■ TECHNICAL STUFF



■ TECHNICAL STUFF

■ LISTING 3-2 vector.py

```
from load import tok, model

paris = tok(" Paris", add_special_tokens=False).input_ids[0]
v = model.model.embed_tokens.weight[paris].tolist()
print("' Paris' is token ID", paris)
print("its list holds", len(v), "numbers")
print("first five:", [round(n, 4) for n in v[:5]])
print("last one: ", round(v[-1], 4))
```

● OUTPUT OF LISTING 3-2

```
$ python3 vector.py
' Paris' is token ID 5242
its list holds 1024 numbers
first five: [-0.0315, 0.042, -0.028, -0.0197, -0.0162]
last one: -0.0183
```

To the model, that list is the word Paris. Every token in its vocabulary has a list of the same length, 1,024 numbers.

A natural first guess is that each number is a field, like a key in a JSON object, so that number 7 might mean "is a city". Nothing supports that guess. Training chose every number, and nobody labelled them. The **parameters** are the model's learned numbers, also called weights. Training sets them. Running the model only reads them. The weights file describes the table only by its name, its number type and its size. It names no single number inside a row. The numbers are not random either. Training kept adjusting them until the model's predictions came out right, and the rest of the chapter shows how the model uses them.

Why lists of numbers at all? Numbers can be multiplied and added, and text cannot. Every step the model takes on a word is arithmetic on its list.

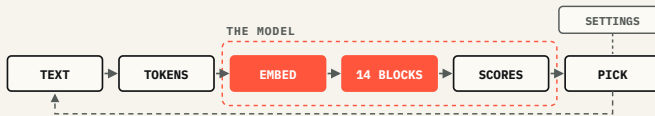
TERM parameters

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



3.2 Why a list of numbers beats the text

**YOU ARE
HERE**
THE TABLE,
THEN THE
BLOCKS



■ WATCH • 1:54



**Why a list of
numbers beats the
text**

Arithmetic is one reason to hold a word as numbers. The bigger reason shows up when you compare two words. A program compares text in two ordinary ways: a keyword search, such as "doctor" in line, and a spelling score that counts the letters equal in the same place. The script below tries both:

■ LISTING 3-3 exact.py

```

notes = ["see a physician today",
         "the doctor is in",
         "he doctored the photos"]
for line in notes:
    print("doctor" in line, " ", line)

def same_letters(a, b):      # letters equal in the same place
    return sum(x == y for x, y in zip(a, b))
for a, b in [("big", "bag"), ("big", "large"),
             ("doctor", "doctrine"), ("doctor", "physician")]:
    print(f"{a} vs {b}: {same_letters(a, b)} letters")
  
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-3

```

$ python3 exact.py
False  see a physician today
True   the doctor is in
True   he doctored the photos
big vs bag: 2 letters
big vs large: 0 letters
doctor vs doctrine: 4 letters
doctor vs physician: 0 letters
  
```



The keyword search misses "see a physician today", which means the same thing with other letters. It catches "he doctored the photos", which shares the letters and means something else. The spelling score puts big nearer bag than large, and doctor nearer doctrine than physician. Text can only be compared by its spelling.

A list of numbers can be close to another list for any reason training found useful: the same meaning, another language, a related place, a nickname. To show it, we need a score for how alike two lists are. **Closeness** is a score for how alike two lists of numbers are. It runs from 1 (alike) through 0 (unrelated) to -1 (opposite). How it is worked out comes after the dot product, a few sections on. In the scripts below, the line marked "even out the sizes" prepares it, and @ then scores the lists against each other.

The script below finds the lists nearest to three words in the model's embedding table. It prints the word's own other spellings first, then the next five:

TERM Closeness

A score for how alike two lists of numbers are. It runs from 1 (alike) through 0 (unrelated) to -1 (opposite).



■ TECHNICAL STUFF

■ LISTING 3-4 near.py

```
import torch
from load import tok, model
torch.set_grad_enabled(False)

E = model.model.embed_tokens.weight.float()
E = E / E.norm(dim=1, keepdim=True)  # even out the sizes
for word in ["Paris", "doctor", "king"]:
    i = tok(word, add_special_tokens=False).input_ids[0]
    close = (E @ E[i]).topk(12).indices.tolist()[1:]
    names = [tok.decode([j]) for j in close]
    same = [n for n in names if n.strip().lower()
            == word.strip().lower()]
    other = [n for n in names if n not in same][:5]
    print(f"{word!r} same word: {' '.join(map(repr, same))}")
    print(f" nearest: {' '.join(map(repr, other))}")
```



● OUTPUT OF LISTING 3-4

```
$ python3 near.py
'Paris' same word: 'Paris' ' paris'
    nearest: 'Paris' ' London' ' Parigi' ' Berlin' ' France'
'doctor' same word: ' Doctor'
    nearest: ' doctors' ' physician' ' Doctors' ' dentist' ' 医師'
'king' same word: ' King' 'King'
    nearest: ' kings' ' queen' ' rey' ' emperor' ' Queen'
```

The model has separate tokens for Paris with and without a capital or a leading space, and training put them next to `Paris`. The script prints them on their own line so nothing is hidden. After them come París and Parigi (Paris in Spanish and Italian), London, Berlin and France. Doctor sits near physician, dentist and 医師, the Japanese word for doctor. King sits near queen, rey (Spanish for king) and emperor. Close means related, not equal: London is near Paris and is another city.

The next script scores the pairs `exact.py` compared by spelling, plus a nickname:

■ LISTING 3-5 pairs.py

```
import torch
from load import tok, model
torch.set_grad_enabled(False)

E = model.model.embed_tokens.weight.float()
E = E / E.norm(dim=1, keepdim=True) # even out the sizes
def row(word):
    return E[tok(word, add_special_tokens=False).input_ids[0]]
for a, b in [(" big", " large"), (" big", " bag"),
             (" doctor", " physician"),
             (" doctor", " doctrine"),
             (" Mike", " Michael"), (" Mike", " Robert")]:
    print(f"{a!r:9} vs {b!r:12} closeness"
          f" {float(row(a) @ row(b)).2f}")
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-5

```
$ python3 pairs.py
'big' vs 'large' closeness 0.55
'big' vs 'bag' closeness 0.14
'doctor' vs 'physician' closeness 0.58
'doctor' vs 'doctrine' closeness 0.13
'Mike' vs 'Michael' closeness 0.43
'Mike' vs 'Robert' closeness 0.22
```

The verdicts reverse. Big is close to large and far from bag. Doctor is close to physician and far from doctrine. Mike sits nearer Michael than Robert. That holds for some nicknames, not all. We probed seven pairs: Dick sat no nearer Richard than another name, and Liz's nearest lists are spelling neighbours such as license. So the lists mostly do not follow spelling, though spelling leaks in now and then.

TERM Context

The other words around a word in the same text.

One more thing text cannot do. **Context** is the other words around a word in the same text. The word bank means one thing next to money and another next to a river. The script below runs four sentences that end in bank. It listens in twice: PyTorch runs a small function each time a layer runs, and these two keep bank's list as it is looked up and as the output layer receives it, after the blocks:



■ TECHNICAL STUFF

■ LISTING 3-6 context.py

```
import torch
from load import tok, model
torch.set_grad_enabled(False)

S = {"money": "I put my money in the bank",
     "account": "He opened an account at the bank",
     "fished": "We fished from the river bank",
     "boat": "The boat drifted toward the bank"}
seen = {} # listen in: first and last step
model.model.embed_tokens.register_forward_hook(
    lambda m, inp, out: seen.update(row=out[0, -1]))
model.lm_head.register_forward_hook(
    lambda m, inp, out: seen.update(h=inp[0][0, -1]))
rows, H = [], []
for s in S.values():
```

CONTINUES OVERLEAF →



context.py CONTINUED

```

model(tok(s, return_tensors="pt").input_ids)
rows.append(seen["row"])    # ' bank' looked up
h = seen["h"].float()      # ' bank' after the blocks
H.append(h / h.norm())     # even out the sizes
print("looked-up ' bank' list same in all four:",
      all(torch.equal(r, rows[0]) for r in rows))
print(" " * 9 + "".join(f"{k:>9}" for k in S))
for k, a in zip(S, H):
    print(f"{k:>9}" + "".join(f"{float(a @ b):9.2f}"
                              for b in H))

```

● OUTPUT OF LISTING 3-6

```

$ python3 context.py
looked-up ' bank' list same in all four: True
      money  account  fished  boat
money      1.00     0.77     0.47  0.48
account    0.77     1.00     0.52  0.53
fished     0.47     0.52     1.00  0.81
boat       0.48     0.53     0.81  1.00

```

The looked-up list for bank is identical in all four sentences. After the blocks it is not. The two money sentences sit together at 0.77, the two river sentences at 0.81, and across the two senses closeness drops to about half, 0.47 to 0.53. The blocks reworked bank's list using its context. That reworked list also carries what the model expects next, so it is not simply "the meaning of bank", and these are four sentences, chosen from six we tried.

That is the design's reason for lists. A string can only equal itself. One list can sit near many words at once, in many ways, and the blocks rework it using the words around it.

CHECK

A keyword search for "doctor" runs over "see a physician today". Does it match? Would the model's lists put the two words close? No, the search misses it. Yes, the lists are close: doctor and physician score 0.58.



■ QUICK CHECK

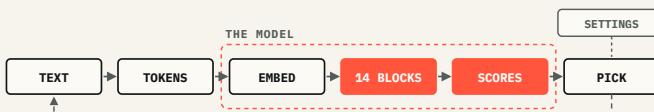
■ WATCH • 1:07



Multiply and add

3.3 Multiply and add: the dot product

YOU ARE
HERE
EVERY
WEIGHT
MATRIX



Comparing two lists starts with a loop you have written many times. Walk both lists together, multiply the two numbers in each spot, and add the product to a running total:



■ TECHNICAL STUFF

■ LISTING 3-7 loop.py

```

a = [2, 0, 1]
b = [3, 5, 4]
total = 0
for i in range(len(a)):
    total += a[i] * b[i] # multiply a pair, add it on
    print(f"i={i}: {a[i]} x {b[i]} = {a[i] * b[i]}")
    f"    total {total}")
print("dot product:", total)
  
```

● OUTPUT OF LISTING 3-7

```

$ python3 loop.py
i=0: 2 x 3 = 6    total 6
i=1: 0 x 5 = 0    total 6
i=2: 1 x 4 = 4    total 10
dot product: 10
  
```

TERM dot product

Multiply two equal-length lists pair by pair, then add up the products. The one number you get scores how well the two lists match.

Two lists went in, and one number came out. That loop has a name. A **dot product** means: multiply two equal-length lists pair by pair, then add up the products. The one number you get scores how well the two lists match. The next section shows why "match" is the right word.

A tempting guess is that multiplying two lists gives a list. Stop after the multiplying and it does: the products here are 6, 0

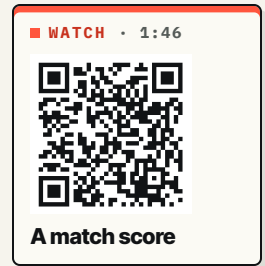
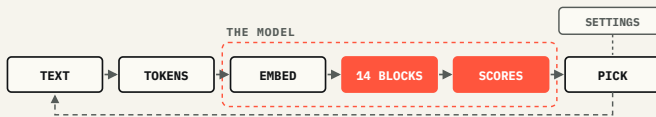


and 4. The dot product keeps going and adds them into a single number, 10.

This loop is worth the attention because the model's weight matrices, its grids of learned numbers, do no other arithmetic. The model has 83 of them, and a later section checks all 83.

3.4 What the number means: a match score

YOU ARE
HERE
EVERY
WEIGHT
MATRIX



A dot product gives one number. To see what that number tells you, run the same loop three times against one list, `q`, and print the products next to each total:

■ LISTING 3-8 meaning.py

```
def dot(a, b):
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    return total

q = [2, -1, 0]
others = {"match": [3, -2, 0], # same signs, same places
         "clash": [-3, 2, 0], # opposite signs
         "ignore": [0, 0, 5]} # big only where q is 0
for name, b in others.items():
    pairs = [q[i] * b[i] for i in range(3)]
    print(f"{name:6} {str(b):11} products {str(pairs):11}"
          f" -> {dot(q, b)}")
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-8

```
$ python3 meaning.py
match [3, -2, 0] products [6, 2, 0] -> 8
clash [-3, 2, 0] products [-6, -2, 0] -> -8
ignore [0, 0, 5] products [0, 0, 0] -> 0
```

The first list has the same signs as q in the same places, and every product is positive. The second has every sign flipped, and every product is negative. The third has a number only where q has a zero, so every product is zero.

Think of each pair as a vote. Two numbers with the same sign vote up. Two numbers with opposite signs vote down. A zero on either side does not vote. Bigger numbers cast bigger votes: in the match, the pair 2 and 3 adds 6, while the pair -1 and -2 adds only 2. The total of the votes is the score. Closeness, from the section on why lists beat text, is this same match score with each list first scaled to the same overall strength, so a list cannot score high only by holding big numbers. The lines marked "even out the sizes" in those scripts do that scaling. The model ranks its next word by the raw match score, not by closeness.

So a dot product is a match score. A large positive score means the two lists rise and fall together, number by number. A large negative score means they pull opposite ways. Zero means the votes balance, so overall they neither agree nor disagree. "Match" here describes the arithmetic only. It does not say two words mean the same thing.

Two wrong readings are common. The first is that the score counts equal numbers. In the match, the only equal pair is the two zeros, which add nothing, and the list still scores 8, because signs and sizes in the same places decide the score. The second is that a score of zero means something failed. Zero means the votes balance. Here no pair votes at all. In longer lists, up votes and down votes can also cancel out. Either way, the two lists neither agree nor disagree overall.



The model uses this because it needs one number for one question. Treat one list as a pattern. The dot product says how strongly another list shows it: strongly, not at all, or the opposite. The model asks that question everywhere, as the next sections show.

CHECK

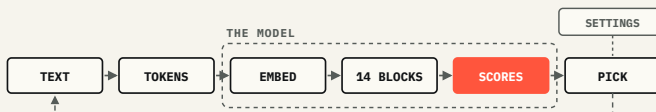
Against $q = [2, -1, 0]$, which scores higher: $[1, -1, 9]$ or $[4, 0, 0]$? $[4, 0, 0]$ scores 8 and $[1, -1, 9]$ scores 3. The 9 is the biggest number in either list, but it sits where q has a zero, so it adds nothing.



■ QUICK CHECK

3.5 The Paris score is one dot product

**YOU ARE
HERE**
THE OUTPUT
LAYER



■ WATCH • 1:49



Paris's score is one dot product

Now the real model. The **output layer** is the last stage. It turns each token's numbers into a score for every token in the vocabulary. The **scores** are 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. Paris got a score of 21 after "The capital of France is". This section shows where that 21 comes from.

The output layer needs a list to work on. It gets one for each token, and the last token's list is the one that picks the next word. That list is not the embedding of "is" any more. A **block** is one repeated stage that reworks every token's numbers. This model stacks 14 of them, and by the last one the list for "is" has been reworked using the whole prompt.

To see that list, we listen in on the output layer. PyTorch lets a script register a small function that runs every time a layer runs. The function below receives whatever the output layer is

TERM output layer

The last stage. It turns each token's numbers into a score for every token in the vocabulary.

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.

TERM block

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



given and keeps it in `seen`. Then the script takes Paris's row of the output table, multiplies the two lists pair by pair, and adds up the products:



■ TECHNICAL STUFF

■ LISTING 3-9 dot.py

```
import torch
from load import tok, model
torch.set_grad_enabled(False) # we only read

P = "The capital of France is"
ids = tok(P, return_tensors="pt").input_ids
seen = {} # what the output layer receives
model.lm_head.register_forward_hook(
    lambda m, inp, out: seen.update(h=inp[0]))
with torch.no_grad():
    logits = model(ids).logits
h = seen["h"][0, -1] # last position: 1,024 numbers
paris = tok(" Paris", add_special_tokens=False).input_ids[0]
row = model.lm_head.weight[paris] # Paris's row: 1,024 numbers
print("h", tuple(h.shape), " row", tuple(row.shape),
      " id", paris)
for a, b in zip(h[:3].tolist(), row[:3].tolist()):
    print(f" {a:8.4f} x {b:8.4f} = {a * b:7.4f}")
print(" ... 1,021 more pairs")
score = (h * row).sum() # multiply pairs, add them up
print("our dot product: ", float(score))
print("model's Paris score:", float(logits[0, -1, paris]))
```

● OUTPUT OF LISTING 3-9

```
$ python3 dot.py
h (1024,) row (1024,) id 5242
-0.5469 x -0.0315 = 0.0172
4.5938 x 0.0420 = 0.1929
-0.8633 x -0.0280 = 0.0241
... 1,021 more pairs
our dot product: 21.0
model's Paris score: 21.0
```

A few lines need reading. The two lines that say "we only read" and `no_grad` tell PyTorch we will not train, so it keeps no extra bookkeeping. `seen["h"][0, -1]` picks the first prompt and its last token, as chapter 2 showed for slicing, so `h` is one list of



1,024 numbers. `(h * row).sum()` is `loop.py` in one line: multiply every pair, then add them all.

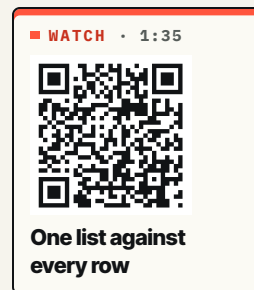
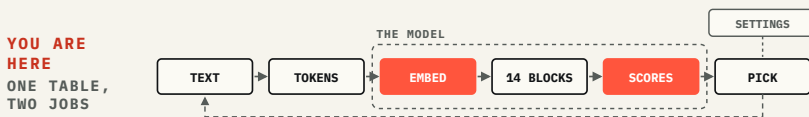
The script prints the first three pairs and their products. All 1,024 products add up to 21.0, and the model's own score for `Paris` is 21.0.



Look at Paris's row. Its first three numbers, -0.0315, 0.0420 and -0.0280, are the first three numbers `vector.py` printed for the word Paris. The output table and the embedding table are one table, used for two jobs. The next section confirms it for the whole table.

So on this prompt, the model's score for Paris is the dot product of these two lists of 1,024 numbers. Training shaped the blocks so the last token's list matches the list of the word that should come next. The closer the match with Paris's list, the higher Paris scores. A single match could be luck on one row. The next section checks all 65,536 rows at once.

3.6 A matrix: one list against every row



The model scores 65,536 tokens, so it needs 65,536 dot products. In code, that is a list of lists and a nested loop. The outer loop walks the rows, and the inner loop is the dot product. The script below stacks the three lists from `meaning.py` as rows. It runs the same list as `q`, here called `x`, against each row:





■ TECHNICAL STUFF

■ LISTING 3-10 matrix.py

```
x = [2, -1, 0]
W = [[-3, 2, 0],      # row 0
      [0, 0, 5],      # row 1
      [3, -2, 0]]     # row 2
scores = []
for row in W:          # one dot product per row
    total = 0
    for i in range(len(x)):
        total += row[i] * x[i]
    scores.append(total)
print("scores:", scores)
best = scores.index(max(scores))
print("best match: row", best)
```

● OUTPUT OF LISTING 3-10

```
$ python3 matrix.py
scores: [-8, 0, 8]
best match: row 2
```

TERM matrix

A grid of numbers, stored as an array of arrays. Each inner array is one row.

TERM row

One inner array of a matrix. In a weight matrix, each row is as long as the input and makes one output number.

TERM Matrix multiplication

Running a list against every row of a matrix, one dot product per row: a nested loop. With several lists, every list meets every row.

Each row is one candidate. Scoring the list against every row ranks the candidates, and the best match wins. Row 2 is the list whose signs agree with x , and it scores highest.

Math has names for each part. A **matrix** is a grid of numbers, stored as an array of arrays. Each inner array is one row. A **row** is one inner array of a matrix. In a weight matrix, each row is as long as the input and makes one output number. **Matrix multiplication** is running a list against every row of a matrix, one dot product per row: a nested loop. With several lists, every list meets every row. In this loop the grid comes first, and the list meets each row. A later section shows what changes when the list comes first.

The model's output layer is one matrix. It has a row for every token in the vocabulary: 65,536 rows of 1,024 numbers. One pass of the nested loop scores every possible next word:



■ LISTING 3-11 rank.py

```

import torch
from load import tok, model
torch.set_grad_enabled(False) # we only read

P = "The capital of France is"
ids = tok(P, return_tensors="pt").input_ids
seen = {}
model.lm_head.register_forward_hook(
    lambda m, inp, out: seen.update(h=inp[0]))
with torch.no_grad():
    logits = model(ids).logits
h = seen["h"][0, -1] # (1024,)
E = model.lm_head.weight # (65536, 1024): a row per token
scores = h @ E.T # 65,536 dot products at once
print("h", tuple(h.shape), "@ E.T", tuple(E.T.shape),
      "->", tuple(scores.shape))
print("equals the model's scores:",
      torch.equal(scores, logits[0, -1]))
emb = model.model.embed_tokens.weight
print("same table as the embeddings:",
      E.data_ptr() == emb.data_ptr())
for s, i in zip(*torch.topk(scores.float(), 5)):
    t = tok.decode([int(i)])
    print(f" {t!r:14} row {int(i):>5} score {float(s):5.2f}")

```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-11

```

$ python3 rank.py
h (1024,) @ E.T (1024, 65536) -> (65536,)
equals the model's scores: True
same table as the embeddings: True
' Paris'      row 5242  score 21.00
' Re'         row 1416  score 15.88
' Strasbourg' row 51596  score 15.69
' Bordeaux'   row 40642  score 15.31
' Lyon'       row 26969  score 15.12

```

In PyTorch, `@` is matrix multiplication, so `h @ E.T` runs the whole nested loop in one line. The `.T` reads the table a particular way, and a later section explains why the code needs it. For now, read the line as "score `h` against every row of `E`". It gives 65,536 numbers. `torch.equal` compares every one with the



model's own scores and reports `True`. The match is exact, so the Paris result was no luck.

The second check confirms that the output table and the embedding table share the same storage. When `Paris` is an input, the model reads row 5242 as its list, a lookup with no multiplying. When `Paris` is a candidate next word, the same row gets a dot product.

The top five are the five tokens chapter 1 showed as probabilities, in the same order. **Softmax** is the step that turns scores into probabilities that are all positive and add up to one. It does not change the order. So the model picks its next word with no special logic. `Paris` wins because its row is the best match for `h` among 65,536 rows.



■ QUICK CHECK

CHECK

The output layer has 65,536 rows. How many dot products does it take to score one token, and how long is each? It takes 65,536 dot products, one per row, and each one pairs up 1,024 numbers.

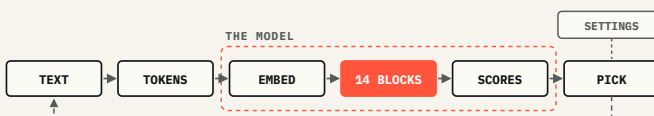
■ WATCH • 1:51



A matrix makes a new list

3.7 A matrix turns one list into another

YOU ARE
HERE
A BLOCK'S
MATRICES



In the output layer, each row's answer is a score to rank. Keep every row's answer instead, in order, and the nested loop has turned one list into a new list. The script below uses two rows, and each one asks the input a question:



■ LISTING 3-12 questions.py

```
def dot(a, b):
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    return total

W = [[1, 1, 1],      # row 0 asks: how much in total?
      [1, 0, -1]]    # row 1 asks: is the first above the last?
x = [4, 1, 3]
y = [dot(row, x) for row in W] # one answer per row
print(len(x), "numbers in: ", x)
print(len(y), "numbers out:", y)
```

● OUTPUT OF LISTING 3-12

```
$ python3 questions.py
3 numbers in:  [4, 1, 3]
2 numbers out: [8, 1]
```



■ TECHNICAL STUFF

Row 0 is all ones, so its dot product adds the input up: 8 in total. Row 1 counts the first number up, ignores the middle one and counts the last one down. Its answer is positive when the first number is bigger than the last, and here it is bigger by one. Three numbers went in, and two came out. So each number of the new list is one row's match score: how strongly the input shows that row's pattern.

A **projection** is a weight matrix turning one list into a new list. Each row asks one question of the input, and each answer is one number of the new list. Two consequences follow. The number of rows sets the length of the new list. And every row must be as long as the input, because each row meets the whole input in a dot product.

The model does this in every block, to rework each token's list into a new one. 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. Picture block 0's first feed-forward matrix, `w1`, as 2,560 strips of 1,024 numbers each. The **feed-forward** part is the part of a block that reworks each

TERM projection

A weight matrix turning one list into a new list. Each row asks one question of the input, and each answer is one number of the new list.

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.



token's numbers on its own, widening them and narrowing back. `w1` does the widening: 1,024 numbers in, 2,560 out, one per row.

The example's rows ask questions you can read. The model's 2,560 rows were set by training, and nobody can read a question off them.

The script below listens in on `w1` during a real run and keeps both its input `x` and its output `y`. Then it works out three output numbers by hand, one row at a time:



■ TECHNICAL STUFF

■ LISTING 3-13 rows.py

```
import torch
from load import tok, model

P = "The capital of France is"
ids = tok(P, return_tensors="pt").input_ids
w1 = model.model.layers[0].feed_forward.w1
seen = {}
w1.register_forward_hook(
    lambda m, inp, out: seen.update(x=inp[0], y=out))
with torch.no_grad():
    model(ids)
x, y, W = seen["x"], seen["y"], w1.weight
print("x", tuple(x.shape), " W", tuple(W.shape),
      " y", tuple(y.shape))
with torch.no_grad():
    print("x @ W.T equals the layer's output:",
          torch.equal(x @ W.T, y))
v = x[0, -1] # the last token's 1,024 numbers
for j in (0, 1, 2559):
    d = W[j] @ v # row j of W, dotted with v
    print(f" output {j:>4}: row {j:>4} . v = "
          f" {float(d):7.4f} layer gave "
          f" {float(y[0, -1, j]):7.4f}")
```

● OUTPUT OF LISTING 3-13

```
$ python3 rows.py
x (1, 6, 1024) W (2560, 1024) y (1, 6, 2560)
x @ W.T equals the layer's output: True
output 0: row 0 . v = 0.3555 layer gave 0.3555
output 1: row 1 . v = -0.3008 layer gave -0.3008
output 2559: row 2559 . v = 0.0962 layer gave 0.0962
```

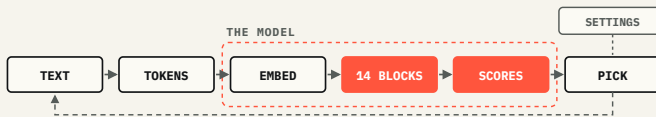


The first line gives sizes. x is one prompt of six tokens with 1,024 numbers each. w is 2,560 rows of 1,024. y is the same six tokens with 2,560 numbers each. Between two lists, $@$ is a plain dot product, so $w[j] @ v$ is row j matched against the last token's list. Output 0 is row 0's answer, output 1 is row 1's, and output 2559 is the last row's. We checked all 2,560 the same way, and every one matched. The second line runs every row for every token at once and matches the whole layer.

Three wrong guesses fail here. The first is that the output is as long as the input. It is not: three in and two out in the example, 1,024 in and 2,560 out in w_1 . The second is that a projection shrinks things, and w_1 widens. The third is that matrix multiplication multiplies matching cells. Output 0 uses all 1,024 numbers of the token, each paired with one number of row 0.

3.8 Every weight matrix runs the same nested loop

YOU ARE
HERE
EVERY
WEIGHT
MATRIX



w_1 is one matrix in one block. The model has 83 weight matrices. Each of its 8 conv blocks has five, each of its 6 attention blocks has seven, and the output layer is one more. Conv and attention blocks are the model's two kinds of block. They mix tokens in different ways, and chapters 8 and 10 open them. Model code gives them different names. The script below finds every one, listens in on all of them while the France prompt runs once, and checks each output against its own nested loop:

WATCH • 1:20



Every weight matrix
runs the same
nested loop





■ TECHNICAL STUFF

■ LISTING 3-14 project.py

```
import torch
from load import tok, model

P = "The capital of France is"
ids = tok(P, return_tensors="pt").input_ids
names = {m: n.removeprefix("model.")
          for n, m in model.named_modules()
          if isinstance(m, torch.nn.Linear)}
seen = []                                     # (matrix, input, output)
for m in names:
    m.register_forward_hook(
        lambda m, inp, out: seen.append((m, inp[0], out)))
with torch.no_grad():
    model(ids)
    same = sum(torch.equal(x @ m.weight.T, y)
                for m, x, y in seen)
print(same, "of", len(seen),
      "weight matrices: output == x @ W.T")
print("biases added:", sum(m.bias is not None for m in names))
for m, x, y in seen:
    n, a, b = names[m], x.shape[-1], y.shape[-1]
    if n.startswith("layers.2.") or n == "lm_head":
        print(f" {n:28} {a:>5} in -> {b:>5} out")
```

● OUTPUT OF LISTING 3-14

```
$ python3 project.py
83 of 83 weight matrices: output == x @ W.T
biases added: 0
layers.2.self_attn.q_proj      1024 in -> 1024 out
layers.2.self_attn.k_proj      1024 in -> 512 out
layers.2.self_attn.v_proj      1024 in -> 512 out
layers.2.self_attn.out_proj     1024 in -> 1024 out
layers.2.feed_forward.w1       1024 in -> 2560 out
layers.2.feed_forward.w3       1024 in -> 2560 out
layers.2.feed_forward.w2       2560 in -> 1024 out
lm_head                       1024 in -> 65536 out
```

PyTorch's name for a weight-matrix layer is `nn.Linear`. All 83 give exactly the nested loop's answer. The second line checks that nothing is added after the multiplying, and in this model nothing is.



The names differ, and so do the lengths. In block 2, two matrices keep 1,024 numbers, two narrow to 512, two widen to 2,560, and one brings 2,560 back to 1,024. The output layer goes from 1,024 to 65,536, one score per token. What the query, key and value matrices are for is chapter 8's subject. Here they are three more grids running the same nested loop.

So on a short prompt like this one, one operation, repeated, does almost all of this model's arithmetic. The model holds more than weight matrices, though. Smaller steps sit between them, such as the norms that rescale each list and the parts that mix tokens together. Every weight matrix is the nested loop from `matrix.py`.

CHECK

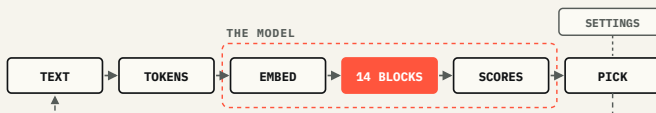
Block 2's `k_proj` is stored as 512 rows of 1,024. How many numbers come out for each token? 512, one per row.



■ QUICK CHECK

3.9 Several lists at once: a batch

**YOU ARE
HERE**
A BLOCK'S
MATRICES



■ WATCH • 1:36



More lists, same
rows

Every run so far sent six tokens through each matrix at once. Sending several lists through the same rows takes one more outer loop, over the lists. The rows stay the same. The script below sends two lists through the two question rows from `questions.py`, then runs the second list again on its own:

■ LISTING 3-15 `twolists.py`

```
def dot(a, b):
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    return total
```

CONTINUES OVERLEAF →



■ TECHNICAL STUFF



twolists.py CONTINUED

```

W = [[1, 1, 1],
      [1, 0, -1]]
xs = [[4, 1, 3],    # list 0
      [0, 2, 5]]    # list 1
both = [[dot(row, x) for row in W] for x in xs]
print("both at once:", both)
alone = [dot(row, xs[1]) for row in W]
print("list 1 alone:", alone, " same:", alone == both[1])

```

● OUTPUT OF LISTING 3-15

```

$ python3 twolists.py
both at once: [[8, 1], [7, -5]]
list 1 alone: [7, -5]  same: True

```

Each answer uses only its own list and the shared rows. So lists never mix inside a weight matrix, and you can stack as many as you like.

TERM batch

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

A **batch** is several independent prompts stacked into one array, so one call handles them all. One call applies the same weights to every token of every prompt, and that is why serving software groups requests into batches. The script below sends France and Spain in one call. It listens in on block 0's `w1`, runs the whole batch through it once, and then runs every token's list alone:



■ TECHNICAL STUFF

■ LISTING 3-16 batch.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
w1 = model.model.layers[0].feed_forward.w1
seen = {}
w1.register_forward_hook(
    lambda m, inp, out: seen.update(x=inp[0]))
with torch.no_grad():
    model(ids)
x, W = seen["x"], w1.weight    # x: batch, position, width

```



batch.py CONTINUED

```

with torch.no_grad():
    batch = x @ W.T          # every row at once
    rows = batch.reshape(12, 2560)
    alone = [x[b, t] @ W.T for b in range(2) for t in range(6)]
    flat = x.reshape(12, 1024) @ W.T  # the same 12 rows
    print("x", tuple(x.shape), "@ W.T", tuple(W.T.shape),
          "->", tuple(batch.shape))
    print("each row alone == its row in the batch:",
          all(torch.equal(a, r) for a, r in zip(alone, rows)))
    print("flat", tuple(flat.shape), "== batch:",
          torch.equal(flat, rows))

```

● OUTPUT OF LISTING 3-16

```

$ python3 batch.py
x (2, 6, 1024) @ W.T (1024, 2560) -> (2, 6, 2560)
each row alone == its row in the batch: True
flat (12, 2560) == batch: True

```

Two prompts of six tokens make 12 lists of 1,024 numbers. The output says "row" for each token's list, because the library stacks the 12 lists as the rows of one array. These are input lists, not rows of `w1`. Each of the 12, run alone, gives exactly its answer from the batch. Laid out as one stack of 12 lists, they give the same answers again. On this laptop, the results are identical.

Inside a weight matrix, France never leaks into Spain. Other parts of a block do mix tokens on purpose: attention and the conv block exist to do that, and chapters 8 and 10 open them. Whether batching is faster, and by how much, is a timing question. Nothing in this chapter was timed.

CHECK

You change only the Spain prompt. Which of `w1`'s 12 answer lists can change? Only Spain's six. Lists never mix inside a weight matrix, and France's six input lists did not change.



■ QUICK CHECK

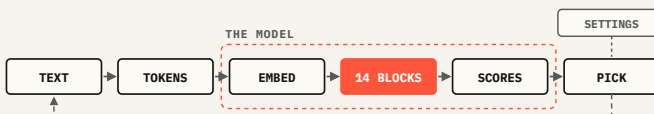
■ WATCH • 1:44



The lengths must
line up

3.10 The lengths must line up

YOU ARE
HERE
A BLOCK'S
MATRICES



The loop pairs numbers one to one. It reads `b[i]` for every `i` of `a`, so if `b` is shorter, there is no `b[i]` to read. The script below tries a list of three against a list of two, first with the loop and then with Python's `zip`:

■ LISTING 3-17 mismatch.py

```
def dot(a, b):
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    return total

a = [2, 0, 1]          # 3 numbers
b = [3, 5]             # 2 numbers
try:
    print(dot(a, b))
except IndexError as e:
    print("IndexError:", e)
pairs = list(zip(a, b)) # zip stops early, silently
print("zip made", len(pairs), "pairs of 3:", pairs)
```

● OUTPUT OF LISTING 3-17

```
$ python3 mismatch.py
IndexError: list index out of range
zip made 2 pairs of 3: [(2, 3), (0, 5)]
```



■ TECHNICAL STUFF



The loop stops with an error. `zip` makes two pairs and raises nothing, so the third number is dropped without a message. A quiet wrong answer is worse than a crash. That is why the library checks lengths before it multiplies. A dot product pairs



numbers one to one, so whatever a list meets must be as long as the list.

Every matrix in the model expects lists of one fixed length, 1,024 or 2,560. The script below tries four multiplications on block o's real feed-forward matrices, `w1` and the narrowing matrix `w2`. The input is all zeros, because only the lengths matter here:

■ LISTING 3-18 match.py

```
import textwrap, torch
from load import model

ff = model.model.layers[0].feed_forward
w1 = ff.w1.weight          # (2560, 1024)
w2 = ff.w2.weight          # (1024, 2560)
x = torch.zeros(2, 6, 1024, dtype=torch.bfloat16)
tries = {"x @ w1.T": lambda: x @ w1.T,
        "x @ w1.T @ w2.T": lambda: x @ w1.T @ w2.T,
        "x @ w1": lambda: x @ w1,
        "x @ w1.T @ w1.T": lambda: x @ w1.T @ w1.T}
for label, f in tries.items():
    try:
        print(f"{label:16} -> {tuple(f().shape)}")
    except RuntimeError as e:
        print(f"{label:16} -> RuntimeError:")
        print(textwrap.fill(str(e), 58, initial_indent="    ",
                           subsequent_indent="    "))
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-18

```
$ python3 match.py
x @ w1.T          -> (2, 6, 2560)
x @ w1.T @ w2.T   -> (2, 6, 1024)
x @ w1            -> RuntimeError:
    mat1 and mat2 shapes cannot be multiplied (12x1024 and
    2560x1024)
x @ w1.T @ w1.T    -> RuntimeError:
    mat1 and mat2 shapes cannot be multiplied (12x2560 and
    1024x2560)
```

The first two tries read each matrix with `.T`, and two sections on explain them. Look at the third, which uses `w1` exactly as it is



stored. The error gives two sizes. 12×1024 is the input: 12 lists of 1,024 numbers. 2560×1024 is the matrix as stored: 2,560 rows of 1,024. Each stored row is 1,024 numbers long, the same as each list, so by the rule above this should work. It fails because @, with the grid on its right, pairs each list with a column, not a row. A column is the numbers at the same place in every row, so each column of the stored `w1` runs down all 2,560 rows. The error's two middle sizes say exactly that: 1,024 numbers per list, 2,560 per column. The next two sections show columns and sides on a small grid. The script wraps PyTorch's message at 58 characters to fit the page, and the words are PyTorch's own.

The 12 can look like a bug, since the input held two prompts of six tokens. It is two prompts times six tokens. The library lays every token of every prompt out as one list before it multiplies, the same 12 lists that `batch.py` checked. The last try is the same slip one step later: `w1` used again where `w2` belongs, so 2,560 numbers meet a matrix that expects 1,024.

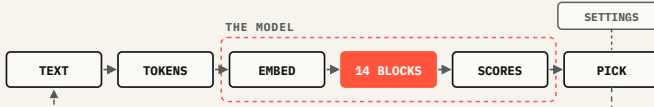
■ WATCH • 0:50



Reading a grid on its side

3.11 Reading a grid on its side

YOU ARE
HERE
A BLOCK'S
MATRICES



The stored matrix holds the right numbers and still does not fit. The fix starts with reading a grid a different way. A grid can be read across its rows or down its columns. A **column** is the numbers at the same place in every row, read from top to bottom. The script below reads a grid of two rows of three column by column, and makes each column a row:

TERM column

The numbers at the same place in every row, read from top to bottom.



■ LISTING 3-19 flipgrid.py

```

W = [[1, 2, 3],
      [4, 5, 6]]          # stored: 2 rows of 3
T = []
for c in range(3):        # each column of W ...
    T.append([row[c] for row in W]) # ... becomes a row
print("stored: ", len(W), "rows of", len(W[0]))
for row in W:
    print("    ", row)
print("flipped:", len(T), "rows of", len(T[0]))
for row in T:
    print("    ", row)

```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-19

```

$ python3 flipgrid.py
stored:  2 rows of 3
  [1, 2, 3]
  [4, 5, 6]
flipped: 3 rows of 2
  [1, 4]
  [2, 5]
  [3, 6]

```

The same six numbers come out, read the other way. The **transpose** is the same matrix read on its side: rows become columns and columns become rows. No number changes. Code writes it `.T`. Reading on its side is only a change of direction.

CHECK

A weight matrix is stored as 512 rows of 1,024 numbers. Read on its side, how many rows does it have, and how long is each? It has 1,024 rows of 512 numbers. The numbers are the same, read in the other direction.

TERM **transpose**

The same matrix read on its side: rows become columns and columns become rows. No number changes. Code writes it `.T`.



■ QUICK CHECK



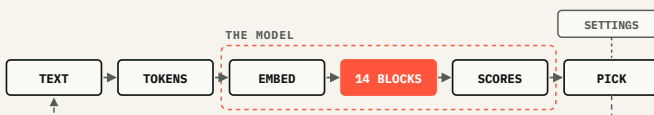
■ WATCH • 1:55



Why the code flips
the stored grid

3.12 Why the code flips the stored grid

YOU ARE
HERE
A BLOCK'S
MATRICES



Flipping changes no number, so why would code ever need it? Because there are two ways to write the same nested loop. Grid first: for each row, pair the list with that row. Every loop in this chapter so far has worked that way. List first: for each column, pair the list with that column. The list's first number pairs with the grid's first row, its second number with the second row, and so on. So each answer runs down one column, and the list must be as long as a column. The script below runs both on the grid from `flipgrid.py`. Then it puts the list first against the same grid read on its side:



■ TECHNICAL STUFF

■ LISTING 3-20 listfirst.py

```
def listfirst(x, G):          # x meets each column of G
    return [sum(x[r] * G[r][c] for r in range(len(G)))
            for c in range(len(G[0]))]

W = [[1, 2, 3],
      [4, 5, 6]]             # stored: 2 rows of 3
a = [1, 0, 1]
rows = [sum(row[i] * a[i] for i in range(3)) for row in W]
print("grid first, [1, 0, 1] meets rows: ", rows)
print("list first, [1, 1] meets columns:", listfirst([1, 1], W))
T = [[row[c] for row in W] for c in range(3)]  # on its side
print("list first, flipped grid:      ", listfirst(a, T))
```



● OUTPUT OF LISTING 3-20

```
$ python3 listfirst.py
grid first, [1, 0, 1] meets rows: [4, 10]
list first, [1, 1] meets columns: [5, 7, 9]
list first, flipped grid: [4, 10]
```

Grid first, the list `[1, 0, 1]` meets each row and gives two numbers, one per row. List first, the list `[1, 1]`, two numbers because each column holds two, meets each column and gives three numbers, one per column. The last line puts `[1, 0, 1]` first against the flipped grid. The flipped grid's columns are the stored rows, so the list meets the stored rows again and gets `[4, 10]`, the grid-first answer.

So the dot products are the same kind either way. The side decides whether the list meets rows or columns. To get the row answers with the list first, flip the grid.

Two facts about PyTorch make this matter for the model's code. First, PyTorch's `nn.Linear` creates its weight with one row per output number: `w1` is stored as 2,560 rows of 1,024. Second, `@` puts the list first: `x @ W` runs the list down each column of `W`. That list-first form is how math writes it, and PyTorch's `@` keeps it. Code keeps the list first because the lists arrive stacked as rows, one per token, and list first hands each token's answer back in its own row, as `batch.py` showed.

Put the two together. Each column of the stored `w1` runs down all 2,560 rows, so it holds 2,560 numbers, and a list of 1,024 cannot pair with it. That is the error from `match.py`. Read `w1` on its side, as `w1.T`, and its columns are the stored rows, each 1,024 numbers long. Now every list meets every stored row, which is the nested loop we want. That is the last line of `listfirst.py` at full size. So the code for a weight matrix is `x @ W.T`, and PyTorch's own documentation writes the layer the same way.

`rows.py` showed this on the real matrix. Its first line printed `W (2560, 1024)`, the stored layout, and its second printed `x @ W.T equals the layer's output: True`. The first two lines of

`match.py` are the working version of the failed try. `x @ w1.T` gives `(2, 6, 2560)`: two prompts, six tokens, 2,560 numbers each. Adding `@ w2.T`, also read on its side, brings each token back to 1,024.

This layout is PyTorch's. Other frameworks and file formats were not inspected here, and some may store matrices the other way round.



■ QUICK CHECK

CHECK

With the list first, `[1, 1]` against `[[1, 2, 3], [4, 5, 6]]`, how many numbers come out? Three, one per column: `[5, 7, 9]`. The list meets columns, and this grid has three of them.

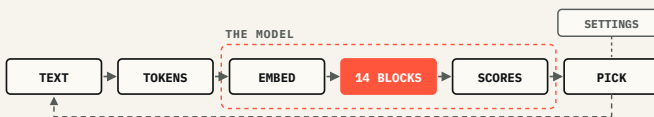
■ WATCH • 1:40



The square trap

3.13 The square trap

**YOU ARE
HERE**
A BLOCK'S
MATRICES



A wrong reading of `w1` stops with an error, because 1,024 and 2,560 differ. A square grid, with as many rows as columns, has the same size both ways round. The script below runs one list through a square grid of two rows of two, first as stored and then read on its side:



■ LISTING 3-21 square.py

```
def apply(W, x):
    # one dot product per row
    return [sum(row[i] * x[i] for i in range(len(x)))
            for row in W]

W = [[1, 2],
      [0, 1]]
T = [[row[c] for row in W] for c in range(2)] # flipped
x = [3, 4]
print("stored: ", W, "->", apply(W, x))
print("flipped:", T, "->", apply(T, x))
print("same size:", len(T) == len(W) and len(T[0]) == len(W[0]),
      " same answer:", apply(T, x) == apply(W, x))
```



■ TECHNICAL STUFF

● OUTPUT OF LISTING 3-21

```
$ python3 square.py
stored: [[1, 2], [0, 1]] -> [11, 4]
flipped: [[1, 0], [2, 1]] -> [3, 10]
same size: True same answer: False
```

Both readings have two rows of two, so nothing raises an error, and the answers differ. A length check compares sizes and nothing else. A square grid has the same size either way round, so the wrong reading passes every length check and computes something else.

The model has 20 square weight matrices of 1,024 by 1,024: the output matrix of each of the 8 conv blocks, and the query and output matrices of each of the 6 attention blocks. Converting weights between formats, or writing the loop by hand, can make exactly this mistake. The script below asks the model for its top three next tokens, stores one square matrix in block o read on its side, and asks again. Storing it flipped has the same effect as code that forgets the `.T`:

■ LISTING 3-22 flip.py

```
import torch
from load import tok, model

P = "The capital of France is"
```

CONTINUES OVERLEAF →



■ TECHNICAL STUFF



flip.py CONTINUED

```

ids = tok(P, return_tensors="pt").input_ids
def top3():
    with torch.no_grad():
        p = torch.softmax(model(ids).logits[0, -1].float(), -1)
        v, i = torch.topk(p, 3)
        return " ".join(f"{tok.decode([int(t)])!r} {float(q):.1%}"
                        for q, t in zip(v, i))

W = model.model.layers[0].conv.out_proj.weight  # (1024, 1024)
print("as stored: ", top3())
with torch.no_grad():
    W.copy_(W.T.clone())  # read it the wrong way round
print("shape now: ", tuple(W.shape))
print("transposed:", top3())

```

● OUTPUT OF LISTING 3-22

```

$ python3 flip.py
as stored:  ' Paris' 95.2%  ' Re' 0.6%  ' Strasbourg' 0.5%
shape now:  (1024, 1024)
transposed:  ' and' 27.7%  ' is' 8.4%  ' in' 7.0%

```

WATCH IT GO WRONG

The model raises no error, the matrix is still 1,024 by 1,024, and the top answer goes from **Paris** at 95.2 percent to **and** at 27.7 percent. Every number is still in the matrix. Each output number now uses a column where it should use a row.

That is one matrix. We also flipped each of the 20 square matrices on its own: 11 kept **Paris** on top with less confidence, 9 changed the top answer, and in every case **Paris** fell below 95.2 percent.

A length check is necessary, and it is not enough. The check that catches this compares numbers. Run your code and a trusted reference on the same real input and compare the outputs. The reference must not share the part you suspect. If you converted the weights, the reference is the original model with its original weights, which is the comparison `flip.py` makes



between its first and last lines. If you wrote the multiplication yourself, the reference is the library's own layer. A third check is to test with matrices whose two sizes differ, so a wrong reading crashes.

CHECK

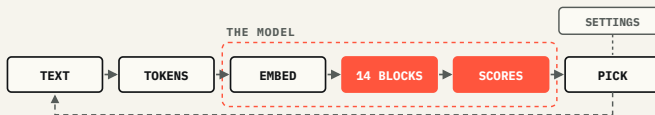
Why did the length check not catch the flipped matrix? It is square. Read either way round, it takes 1,024 numbers and gives 1,024, so both readings have the same size.



■ QUICK CHECK

3.14 Counting the work

**YOU ARE
HERE**
EVERY
WEIGHT
MATRIX



■ WATCH • 1:41



Count the work

Every weight matrix runs the same nested loop, so counting its work means counting how often the innermost line runs. The script below sends two tokens through a grid of three rows of two and counts:

■ LISTING 3-23 count.py

```
W = [[1, 0],
      [0, 1],
      [1, 1]]                # 3 rows of 2: 2 in, 3 out
tokens = [[5, 7], [1, 2]]    # two tokens, 2 numbers each
count = 0
for x in tokens:
    out = []
    for row in W:
        total = 0
        for i in range(len(x)):
            total += row[i] * x[i]
            count += 1         # one multiply-add
        out.append(total)
    print("token", x, "->", out, " multiply-adds:", count)
weights = sum(len(row) for row in W)
```

CONTINUES OVERLEAF →



■ TECHNICAL STUFF



count.py **CONTINUED**

```
print("weights:", weights, " tokens:", len(tokens),
      " multiply-adds:", count)
```

● **OUTPUT** OF LISTING 3-23

```
$ python3 count.py
token [5, 7] -> [5, 7, 12] multiply-adds: 6
token [1, 2] -> [1, 2, 3] multiply-adds: 12
weights: 6 tokens: 2 multiply-adds: 12
```

TERM multiply-add

One multiplication added to a running total. A dot product of 1,024 pairs takes 1,024 of them.

A **multiply-add** is one multiplication added to a running total. A dot product of 1,024 pairs takes 1,024 of them. The grid holds six weights, and each token takes six multiply-adds. Every weight is used exactly once per token, so a weight matrix costs one multiply-add per weight, per token.

The script below adds that up over the model's 83 weight matrices, from their sizes. For each matrix, numbers in times numbers out is the number of weights. It also counts the model's parameters to compare:



■ TECHNICAL STUFF

■ **LISTING 3-24** work.py

```
import torch
from load import model

mats = [m for m in model.modules()
        if isinstance(m, torch.nn.Linear)]
per_token = sum(m.in_features * m.out_features for m in mats)
head = model.lm_head.in_features * model.lm_head.out_features
params = sum(p.numel() for p in model.parameters())
print("weight matrices:      ", len(mats))
print(f"multiply-adds per token: {per_token:,}")
print(f" in the output layer:      {head:,}")
print(f"parameters:                  {params:,}")
```



● OUTPUT OF LISTING 3-24

```
$ python3 work.py
weight matrices:      83
multiply-adds per token: 229,638,144
  in the output layer: 67,108,864
parameters:          229,693,184
```

We counted 229,638,144 multiply-adds per token, about 230 million, from the matrix sizes. The output layer alone takes 67,108,864 of them, 29 percent, because it has 65,536 rows. It is one matrix out of 83 and the largest single share.

The count is almost the parameter count, the number of learned numbers in the model. Nearly every parameter sits in a weight matrix and is used once per token. The 55,040 parameters outside the matrices belong to the norms and the conv filters. The embedding table is counted once as parameters but used twice: read as a lookup at the start, with no multiplying, and multiplied as the output layer at the end.

This count covers the 83 weight matrices only. It leaves out work whose size grows with the prompt, such as attention's own scores and the conv filter, plus the norms, the feed-forward gate and softmax. It is a count of multiply-adds, not a time and not a memory size. Chapter 13 measures cost.

CHECK

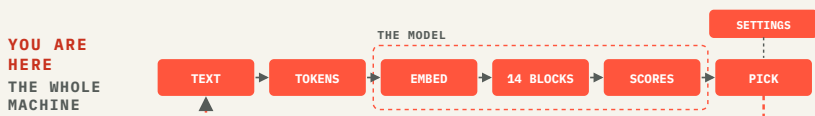
A weight matrix has 4 rows of 3. How many multiply-adds does it take per token? 12, one per weight.



■ QUICK CHECK

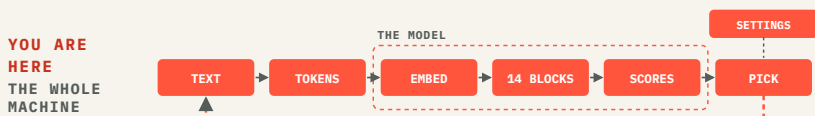


3.15 The map



PART	OPENS IN
Why the Paris sum is 21.0 and not about 21.03, and two-byte numbers	B1C04
How softmax turns scores into probabilities	B1C05
Closeness and other ways to compare lists	B1C05 or later
How training sets the numbers in each row	B1C06
The embedding lookup and positions	B1C07
What queries, keys, values, heads, norms and the feed-forward gate do	B1C08
How the blocks use the words around a word	B1C08 and B1C10
The conv block and its output matrix	B1C10
Time, memory, the cost of attention, and whether batching is faster	B1C13

3.16 Put it together



To the model, a word is a list of 1,024 numbers that training chose and nobody labelled. Words with related meanings have close lists, whatever their spelling, and the blocks rework a word's list with its context. A dot product is a loop over two lists of the same length: multiply each pair and add the products into one number. That number scores how well



the lists match, because same-sign pairs vote up, opposite-sign pairs vote down and zeros do not vote. A score of zero means the votes balance. A matrix is an array of arrays, one row per question. Running a list against every row is a nested loop, and its answers form a new list as long as the number of rows. Every one of this model's 83 weight matrices does exactly that, including the output layer, whose 65,536 rows score every possible next word. `Paris` is on top because its row matches the last token's list best, with a score of 21. More lists only mean more turns of the outer loop, and they never mix. Each row must be as long as the list it meets. The side matters: with the grid first a list meets each row, and with the list first, as PyTorch's `@` writes it, a list meets each column. PyTorch stores one row per output number, so the code reads the stored grid on its side, `x @ W.T`, whose columns are the stored rows. A square grid fits either way round, so only a comparison of numbers catches it being read backwards. And each weight costs one multiply-add per token, about 230 million per token here.

Every one of those numbers is stored in two bytes. Worked out with more precise numbers, the same `Paris` dot product comes to about 21.03, and the model kept 21. The next chapter asks what a number loses when it has only two bytes.

■ Further reading

- ◆ PyTorch documentation: `torch.nn.Linear`, `torch.nn.functional.linear`, and `torch.matmul`.
- ◆ Hugging Face Transformers documentation: the LFM2 model page, and the `modeling_lfm2.py` source for the feed-forward, attention, conv and output-layer matrices.



Cheat Sheet



THE MACHINE • WHERE EACH PART IS TAUGHT

3.1 A word is a list of numbers



3.2 Why a list of numbers beats the text



3.3 Multiply and add: the dot product



3.4 What the number means: a match score



3.5 The Paris score is one dot product



3.6 A matrix: one list against every row



3.7 A matrix turns one list into another



3.8 Every weight matrix runs the same nested loop



3.9 Several lists at once: a batch



3.10 The lengths must line up



3.11 Reading a grid on its side



3.12 Why the code flips the stored grid



3.13 The square trap



3.14 Counting the work



TERMS • 20 IN THIS CHAPTER

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.

token A small piece of text, often a word or part of one, that the model reads and writes one at a time.

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

vector A list of numbers. One token's 1,024 numbers are a vector. In code, a plain array.

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

closeness A score for how alike two lists of numbers are. It runs from 1 (alike) through 0 (unrelated) to -1 (opposite).

context The other words around a word in the same text.

dot product Multiply two equal-length lists pair by pair, then add up the products. The one number you get scores how well the two lists match.

output layer The last stage. It turns each token's numbers into a score for every token in the vocabulary.

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

matrix A grid of numbers, stored as an array of arrays. Each inner array is one row.

row One inner array of a matrix. In a weight matrix, each row is as long as the input and makes one output number.

matrix multiplication Running a list against every row of a matrix, one dot product per row: a nested loop. With several lists, every list meets every row.

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

projection A weight matrix turning one list into a new list. Each row asks one question of the input, and each answer is one number of the new list.

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

column The numbers at the same place in every row, read from top to bottom.

transpose The same matrix read on its side: rows become columns and columns become rows. No number changes. Code writes it .T.

multiply-add One multiplication added to a running total. A dot product of 1,024 pairs takes 1,024 of them.

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

