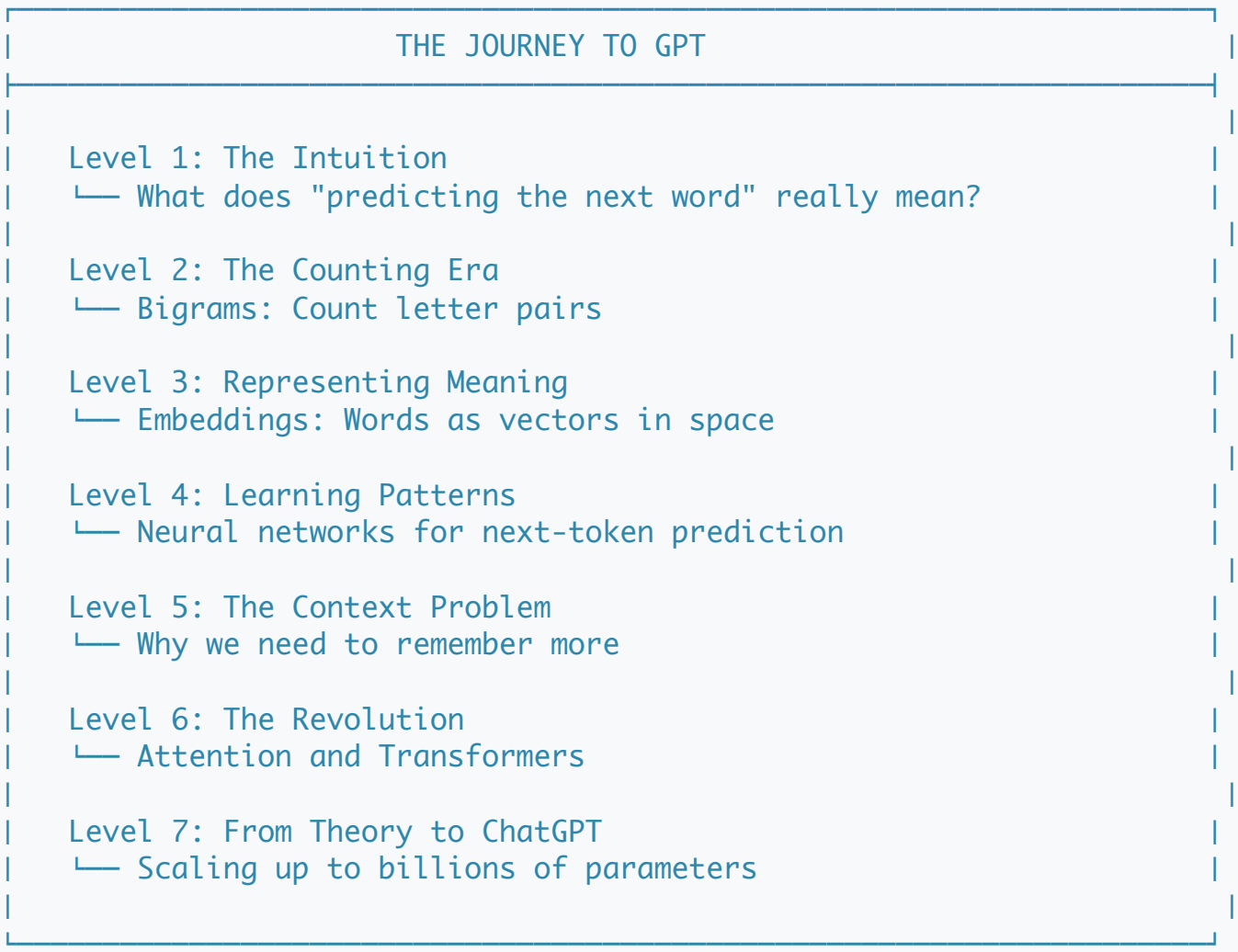
Next Token Prediction

Building ChatGPT from Scratch (Conceptually)

Nipun Batra · IIT Gandhinagar

What We'll Learn Today



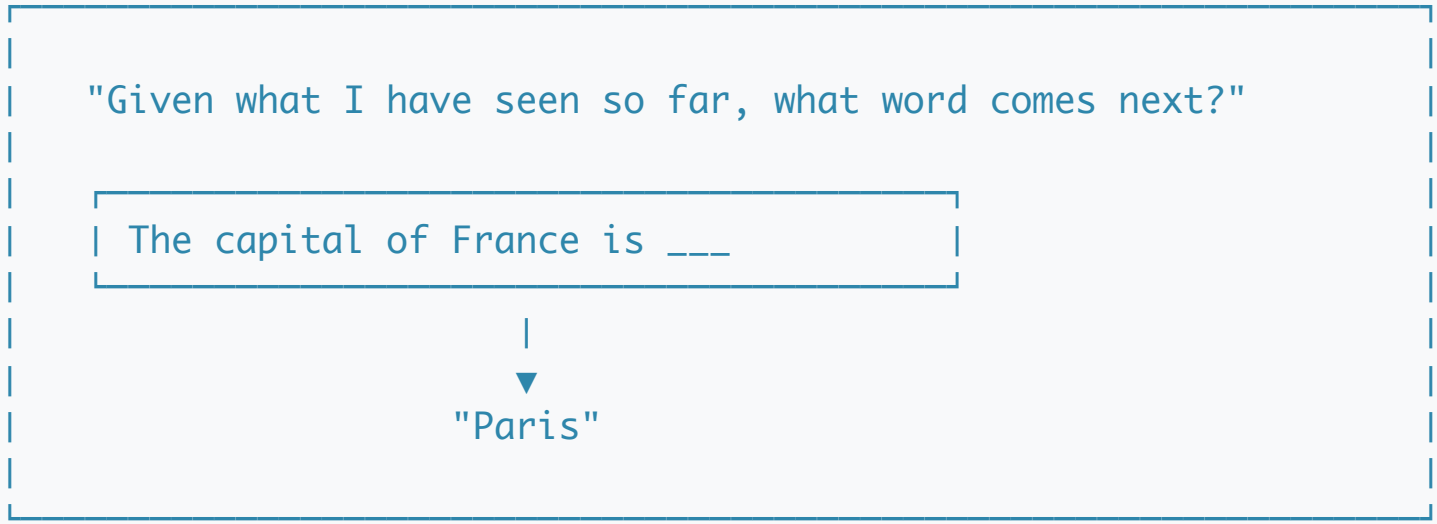
The Journey Ahead

Level 1: The Intuition

What Are We Really Doing?

The Core Problem

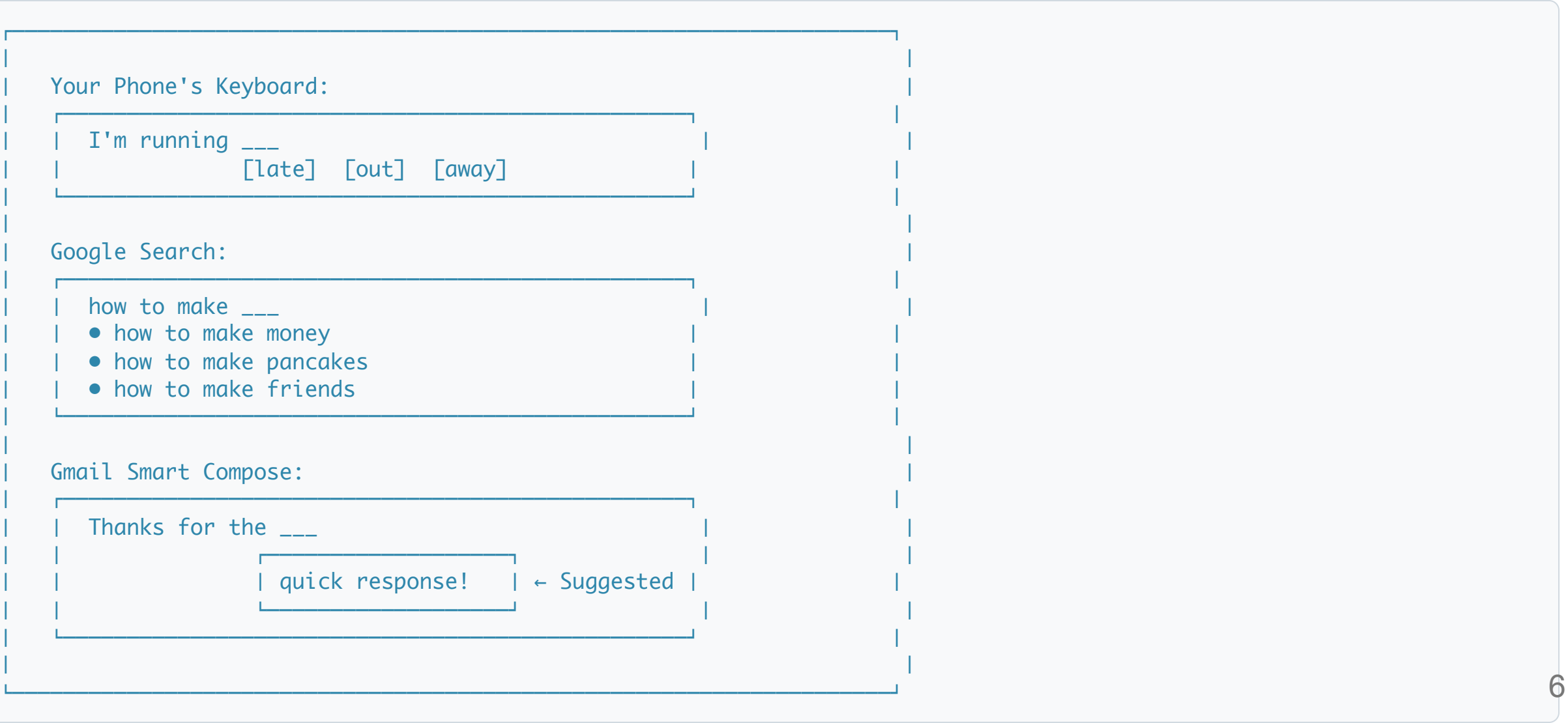
Every language model answers **one simple question**:



That's it. **Predict the next word. Repeat until done.**

You Already Know This!

You've been using next-word prediction your whole life:



Let's Play: The Autocomplete Game

Round 1: "The Eiffel Tower is located in ___"

Your brain: **Paris** (very confident!)

Round 2: "I want to eat ___"

Your brain: **pizza? pasta? nothing?** (uncertain!)

Round 3: "Once upon a ___"

Your brain: **time** (almost certain!)

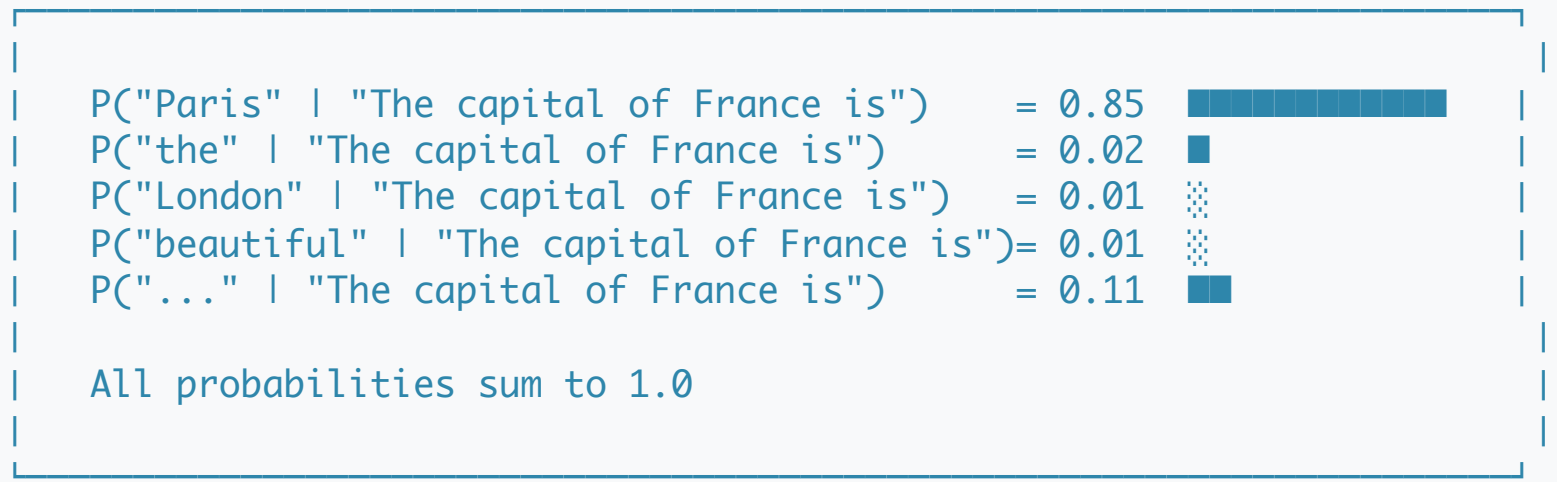
Round 4: "To be or not to ___"

Your brain: **be** (Shakespeare hardcoded in culture!)

Your brain assigns **probabilities** to each possible next word. Some contexts have obvious answers, others don't!

The Mathematical View

When you read "The capital of France is ___", your brain computes:



This is called a PROBABILITY DISTRIBUTION over the vocabulary.
Language models learn to produce these distributions!

It's JUST Prediction

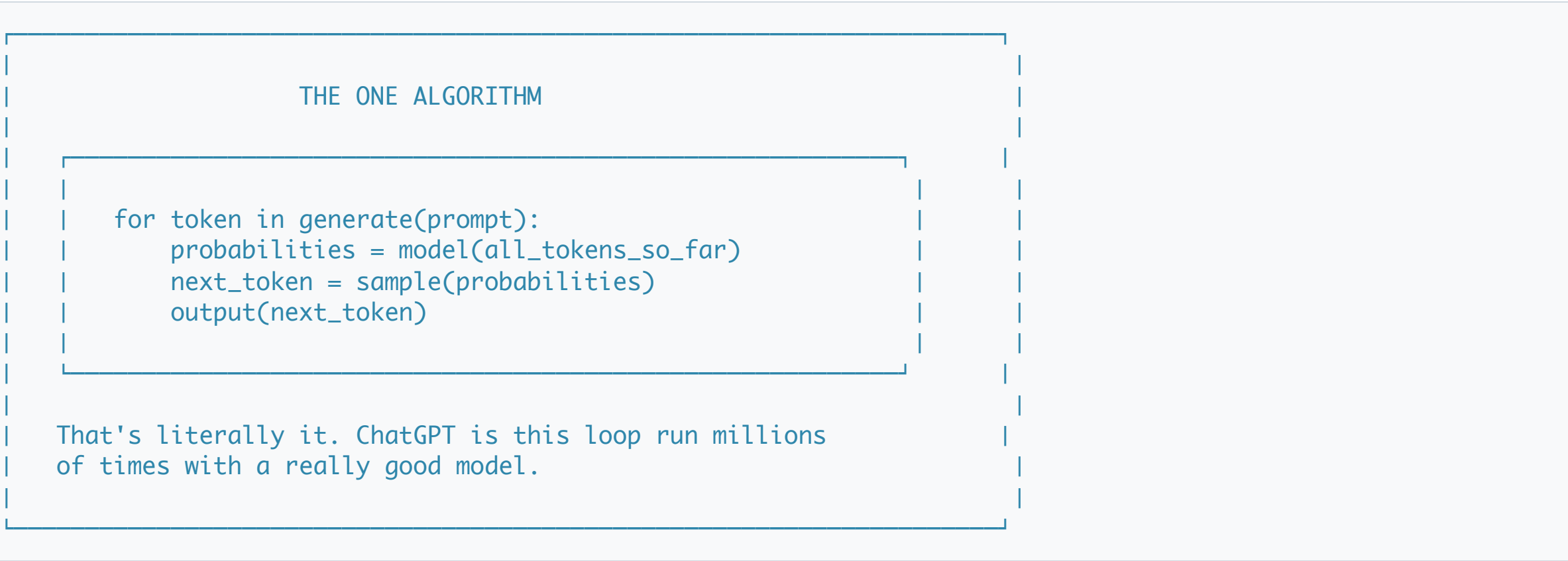
You might think ChatGPT "understands" physics or history.

But all it does is predict the next word.

```
Prompt: "F = m"                 Prediction: "a"      ← Newton's Law!
Prompt: "To be or not to"       Prediction: "be"     ← Shakespeare!
Prompt: "E = mc"                Prediction: "²"      ← Einstein!
Prompt: "print('Hello"          Prediction: "')"     ← Python syntax!
Prompt: "2 + 2 ="               Prediction: "4"      ← Math!
Prompt: "The mitochondria is"   Prediction: "the"    ← Biology meme!
```

> If you predict well enough, you **appear** to understand everything. The model has compressed patterns from human knowledge into its weights.

The Shocking Simplicity



The Magic of "Just Prediction"

```
┌──────────────────────────────────────────────────────────────┐
| Q: "What is 17 + 28?"                                        |
|                                                              |
| The model has seen THOUSANDS of math problems in training:   |
|   "2 + 2 = 4"                                                |
|   "15 + 10 = 25"                                             |
|   "17 + 28 = 45"   ← Saw this pattern!                      |
|                                                              |
| So when asked "17 + 28 =", it predicts "45"                  |
| Not because it "knows" math, but because that pattern exists! |
└──────────────────────────────────────────────────────────────┘
```

This is why LLMs can make math mistakes — they're pattern matching, not actually computing! Try asking "What is 4738 × 2951?" and you'll see errors.

Emergent Behaviors

As models get bigger, surprising abilities **emerge**:

```
┌──────────────────────────────────────────────────────────────────┐
│                    WHAT EMERGES FROM PREDICTION                  │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│   Small Model (100M parameters):                                 │
│   • Complete simple sentences                                    │
│   • Basic grammar                                                │
│                                                                  │
│   Medium Model (1B parameters):                                  │
│   • Answer factual questions                                     │
│   • Simple reasoning                                             │
│                                                                  │
│   Large Model (100B+ parameters):                                │
│   • Complex reasoning                                            │
│   • Code generation                                              │
│   • Creative writing                                             │
│   • Multi-step problem solving                                   │
│   • "Understanding" context and nuance                           │
│                                                                  │
│   All from the same objective: predict the next token!           │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Level 2: The Counting Era

Bigrams: The Simplest Language Model

The Simplest Possible Model

Idea: Just count what letter usually follows each letter.

Training data: Names like `aabid`, `zeel`, `priya`, `nipun`

Count transitions:

- After `a`: saw `a` (1 time), `b` (1 time)
- After `z`: saw `e` (1 time)
- After `e`: saw `e` (1 time), `l` (1 time)
- After `n`: saw `i` (1 time) in "nipun"

This is called a **Bigram** model (looks at pairs of 2 characters).

Let's Build It Step by Step

```
Training Data: "aabid", "priya", "zeel", "nipun"

Step 1: Add special tokens
        ".aabid.", ".priya.", ".zeel.", ".nipun."
        (. marks beginning and end)

Step 2: Count all pairs
        ".a" appears 2 times   (from aabid, priya doesn't start with 'a')
        "aa" appears 1 time
        "ab" appears 1 time
        "bi" appears 1 time
        "id" appears 1 time
        "d." appears 1 time
        ... and so on

Step 3: Convert counts to probabilities
        P(next = 'a' | current = '.') = Count(".a") / Total pairs starting with "."
        P(next = 'a' | current = '.') = 2 / 4 = 0.50
```

Bigram: The Counting Table

```
          Next Character →
        ┌─────┬─────┬─────┬─────┬─────┬─────┐
        |  a  |  b  |  e  |  i  |  l  | ... |
   ┌────┼─────┼─────┼─────┼─────┼─────┼─────┤
C  a   | 0.3 | 0.2 | 0.1 | 0.2 | 0.1 | ... |  (probabilities)
u  b   | 0.1 | 0.0 | 0.1 | 0.5 | 0.0 | ... |
r  e   | 0.2 | 0.0 | 0.3 | 0.1 | 0.2 | ... |
r  i   | 0.4 | 0.1 | 0.1 | 0.0 | 0.1 | ... |
   ↓   | ... | ... | ... | ... | ... | ... |
   └────┴─────┴─────┴─────┴─────┴─────┴─────┘

Each row sums to 1.0 (it's a probability distribution!)

To generate: Look up current letter → Sample from that row
```

This table IS the model. No neural network needed!

Generating Names with Bigrams

```
Step 1: Start with "." (beginning token)
        Look up row "." → High prob for 'a', 's', 'm'
        Sample → Got 'a'

Step 2: Current = 'a'
        Look up row "a" → Moderate prob for 'a', 'b', 'n'
        Sample → Got 'b'

Step 3: Current = 'b'
        Look up row "b" → High prob for 'i', 'a', 'r'
        Sample → Got 'i'

Step 4: Current = 'i'
        Look up row "i" → High prob for 'd', 'n', 'a'
        Sample → Got 'd'

Step 5: Current = 'd'
        Look up row "d" → High prob for "." (end token)
        Sample → Got "."   (DONE!)

Result: "abid"  ← Looks like a real name!
```

Interactive Example: Generating from Bigrams

```
┌──────────────────────────────────────────────────────────────────┐
│                    BIGRAM GENERATION DEMO                        │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  Current: .                                                      │
│  Options: a(0.25), m(0.20), s(0.15), j(0.10), r(0.08), ...       │
│  Rolled: 0.18 → Selected 'm'                                     │
│                                                                  │
│  Current: m                                                      │
│  Options: a(0.40), i(0.25), o(0.15), u(0.10), ...                │
│  Rolled: 0.32 → Selected 'a'                                     │
│                                                                  │
│  Current: a                                                      │
│  Options: n(0.20), r(0.18), l(0.15), y(0.12), .(0.10), ...       │
│  Rolled: 0.45 → Selected 'r'                                     │
│                                                                  │
│  Current: r                                                      │
│  Options: i(0.25), a(0.20), y(0.18), .(0.15), ...                │
│  Rolled: 0.52 → Selected 'y'                                     │
│                                                                  │
│  Current: y                                                      │
│  Options: .(0.45), a(0.15), i(0.10), ...                         │
│  Rolled: 0.30 → Selected '.'                                     │
│                                                                  │
│  Generated: "mary"  ← A real name!                               │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Why Bigrams Fail: The Memory Problem

The Problem:

```
Sentence: "The quick brown
           fox jumps over
           the lazy dog."

Question: After "dog",
          what comes next?

Bigram sees: "dog" → ?
             (forgot everything
              before "dog"!)
```

Context is Lost:

```
With context:
"The cat sat on the ___"
  → Probably "mat"

Without context:
"the ___"
  → Could be anything!

Bigram only sees 1 char!
```

> Bigrams have **no memory**. They forget everything except the last character!

A Concrete Example

```
┌──────────────────────────────────────────────────────────────────────┐
│                    THE CONTEXT PROBLEM                               │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Sentence 1: "I love eating pizza with extra cheese"                │
│   Sentence 2: "I love eating pizza with my friends"                  │
│                                                                      │
│   After "with", what comes next?                                     │
│                                                                      │
│   Bigram's view: "h" → ?                                             │
│   (It doesn't even know it's in "with"!)                             │
│                                                                      │
│   A smarter model would know:                                        │
│   - "pizza with" usually followed by toppings or people              │
│   - "eating with" suggests companions                                │
│   - "love eating" suggests food context                              │
│                                                                      │
│   We need to see MORE context!                                       │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

The Curse of Dimensionality

Why not just count longer patterns?

```
1-gram (Unigram):   27 entries           ← Fits in memory
2-gram (Bigram):    27² = 729            ← Still fine
3-gram (Trigram):   27³ = 19,683         ← OK
4-gram:             27⁴ = 531,441        ← Getting big
5-gram:             27⁵ = 14,348,907     ← Very big
10-gram:            27¹⁰ ≈ 205 TRILLION  ← Impossible!

For words (50,000 vocabulary):
2-gram:             50,000² = 2.5 BILLION
3-gram:             50,000³ = 125 TRILLION
```

> We can't just count longer patterns — we need to **generalize**. This is where neural networks come in!

Bigrams: Summary

Aspect	Bigrams
What it does	Counts P(next char | current char)
Memory	1 character only
Size	27 × 27 = 729 numbers
Speed	Instant (just table lookup)
Quality	Poor (no context)
Training	Just counting

Key insight: The model is just a lookup table. No learning, no generalization.

Level 3: Representing Meaning

Embeddings: Words as Vectors

How Do Computers Read?

Computers only understand numbers. How do we convert letters?

Option A: One-Hot Encoding

```
'a' = [1, 0, 0, 0, ..., 0]    (27 dimensions for letters)
'b' = [0, 1, 0, 0, ..., 0]
'c' = [0, 0, 1, 0, ..., 0]
...
'z' = [0, 0, 0, 0, ..., 1]
```

Problem: These vectors are **orthogonal** (dot product = 0).

The computer thinks 'a' and 'b' are completely unrelated!

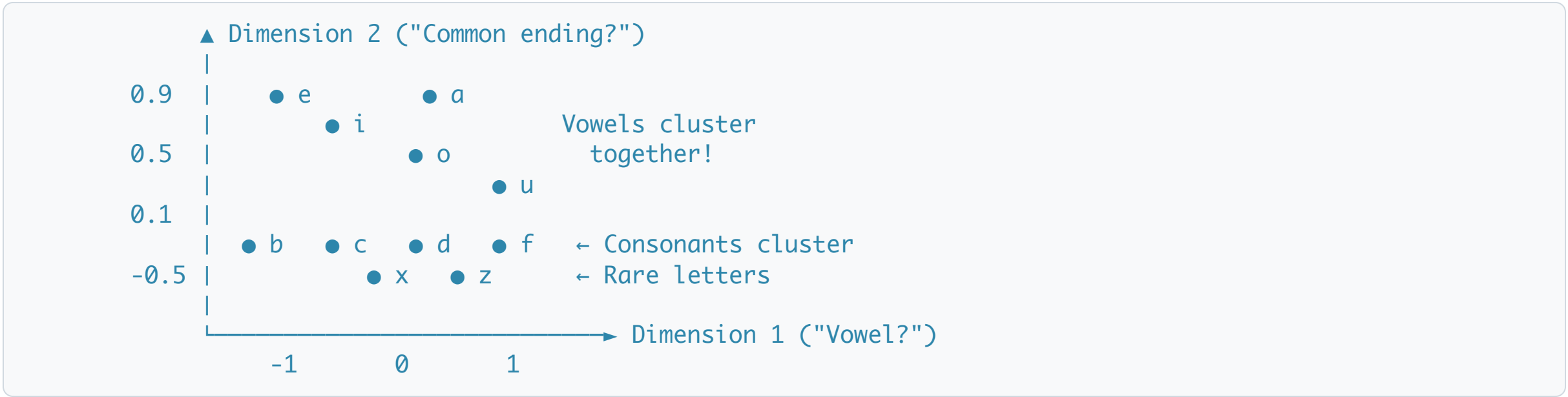
The Problem with One-Hot

```
Distance between letters:

   'a' ●                    'b' ●

   Distance(a, b) = Distance(a, z) = √2

   Every letter is equally far from every other letter!

   But we KNOW:
   - 'a' and 'e' are both vowels (similar!)
   - 'a' and 'x' have nothing in common (different!)
   - 'p' and 'b' look similar (related!)
```

We need a smarter representation where **similar things are close**.

Dense Embeddings: Meaning as Coordinates

Idea: Represent each character as a point in space where **similar things are close**.



Now `a` and `e` are **mathematically close**!

Word Embeddings: The Famous Example

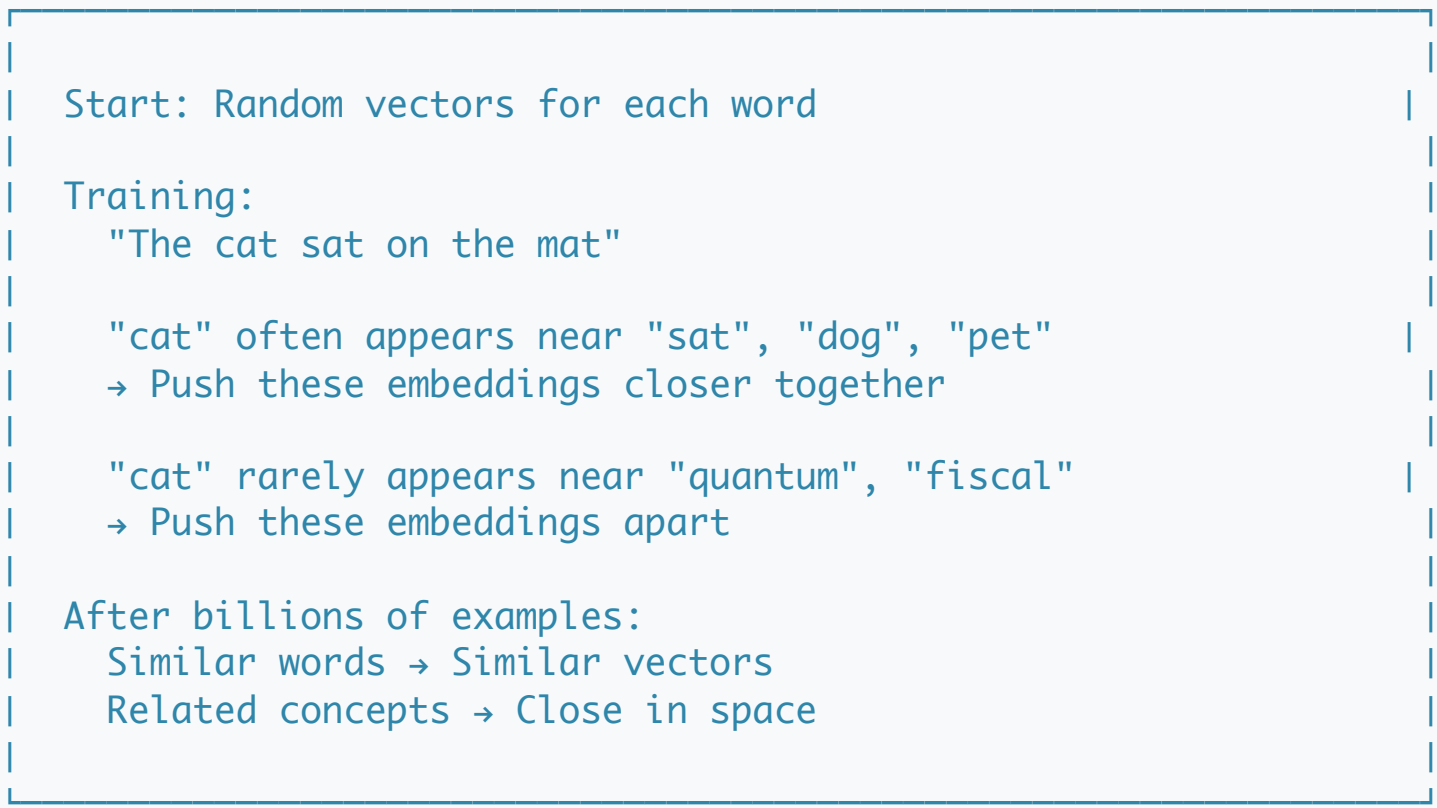
The King - Man + Woman = Queen Example

```
┌──────────────────────────────────────────────────────────────────────┐
│                      WORD ARITHMETIC                                 │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   king  = [0.8, 0.3, 0.9, ...]  (royalty, male, power)               │
│   man   = [0.1, 0.3, 0.5, ...]  (person, male, average)              │
│   woman = [0.1, 0.9, 0.5, ...]  (person, female, average)            │
│                                                                      │
│   king - man + woman = ?                                             │
│                                                                      │
│   [0.8, 0.3, 0.9] - [0.1, 0.3, 0.5] + [0.1, 0.9, 0.5]                │
│   = [0.8, 0.9, 0.9]                                                  │
│                                                                      │
│   Nearest word to [0.8, 0.9, 0.9]: "queen"!                          │
│                                                                      │
│   The model learned that:                                            │
│   "The relationship between king and man is the same as              │
│    the relationship between queen and woman"                         │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

More Word Analogies

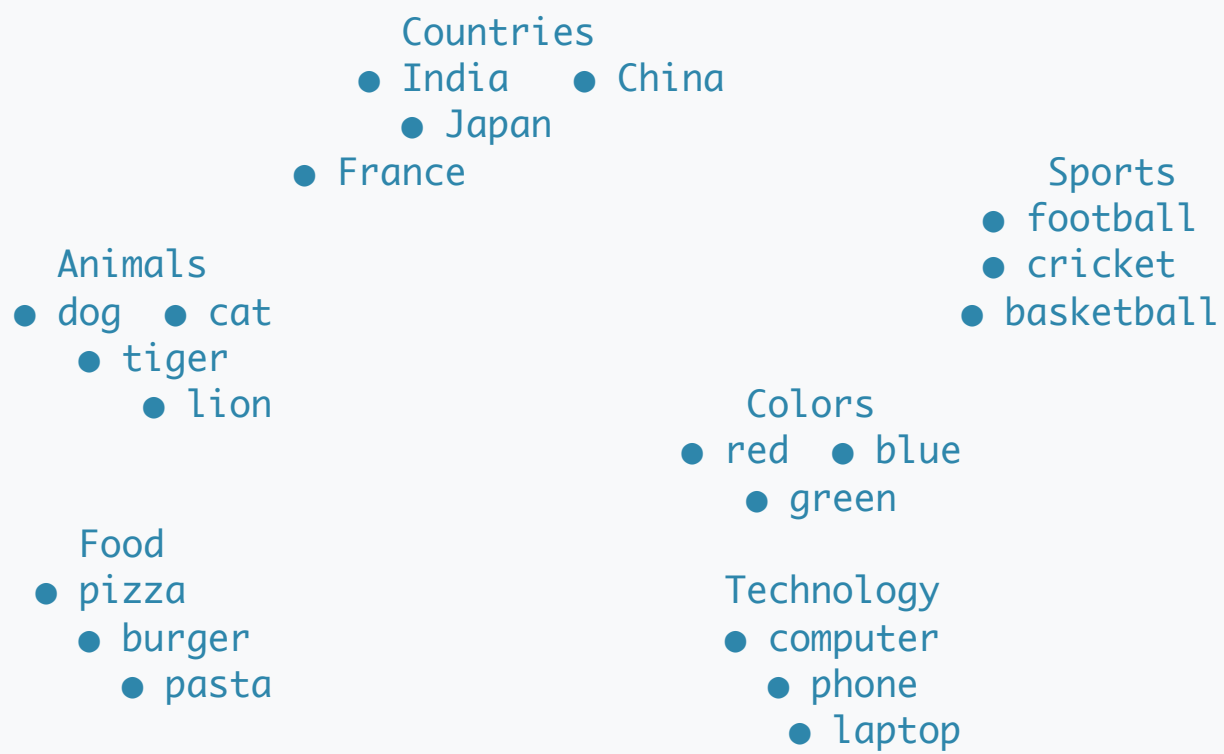
```
┌──────────────────────────────────────────────────────────────────────┐
│                     EMBEDDING ANALOGIES                              │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   France : Paris :: Japan : ?                                        │
│   → Tokyo                                                            │
│                                                                      │
│   good : better :: bad : ?                                           │
│   → worse                                                            │
│                                                                      │
│   walking : walked :: swimming : ?                                   │
│   → swam                                                             │
│                                                                      │
│   Einstein : physicist :: Picasso : ?                                │
│   → painter                                                          │
│                                                                      │
│   The embeddings capture RELATIONSHIPS automatically!                │
│   No one told the model about capitals or verb tenses!               │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

How Embeddings Are Learned



Visualizing Embeddings

Real word embeddings projected to 2D (using t-SNE):



Similar things automatically cluster together!

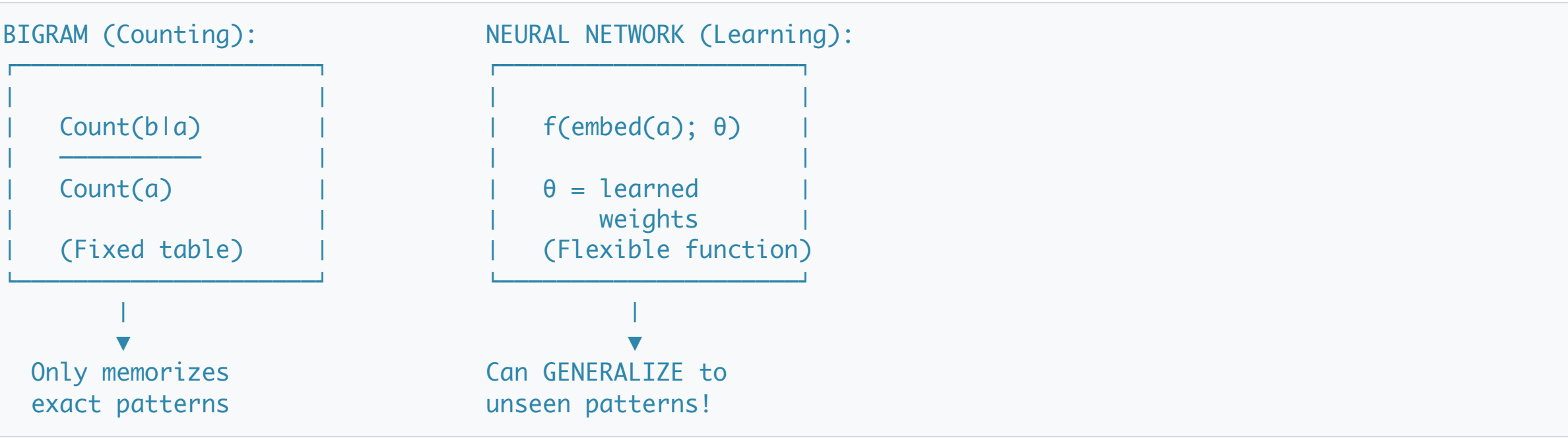
Embedding Dimensions

```
┌──────────────────────────────────────────────────────────────────────┐
│                    WHAT DO DIMENSIONS MEAN?                          │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   We use 256-4096 dimensions in practice, but imagine 4:             │
│                                                                      │
│   Dimension 1: "Is it alive?"                                        │
│   Dimension 2: "Is it a person?"                                     │
│   Dimension 3: "Is it concrete vs abstract?"                         │
│   Dimension 4: "Positive or negative sentiment?"                     │
│                                                                      │
│   "dog"    = [0.9, 0.1, 0.8, 0.7]  (alive, not person, concrete, +)  │
│   "cat"    = [0.9, 0.1, 0.8, 0.6]  (very similar to dog!)            │
│   "love"   = [0.2, 0.3, -0.8, 0.9] (abstract, positive)              │
│   "hate"   = [0.2, 0.3, -0.8, -0.9](abstract, negative)              │
│   "table"  = [0.0, 0.0, 0.9, 0.0]  (not alive, concrete, neutral)    │
│                                                                      │
│   In reality, dimensions are learned and not so interpretable!       │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Level 4: Learning Patterns

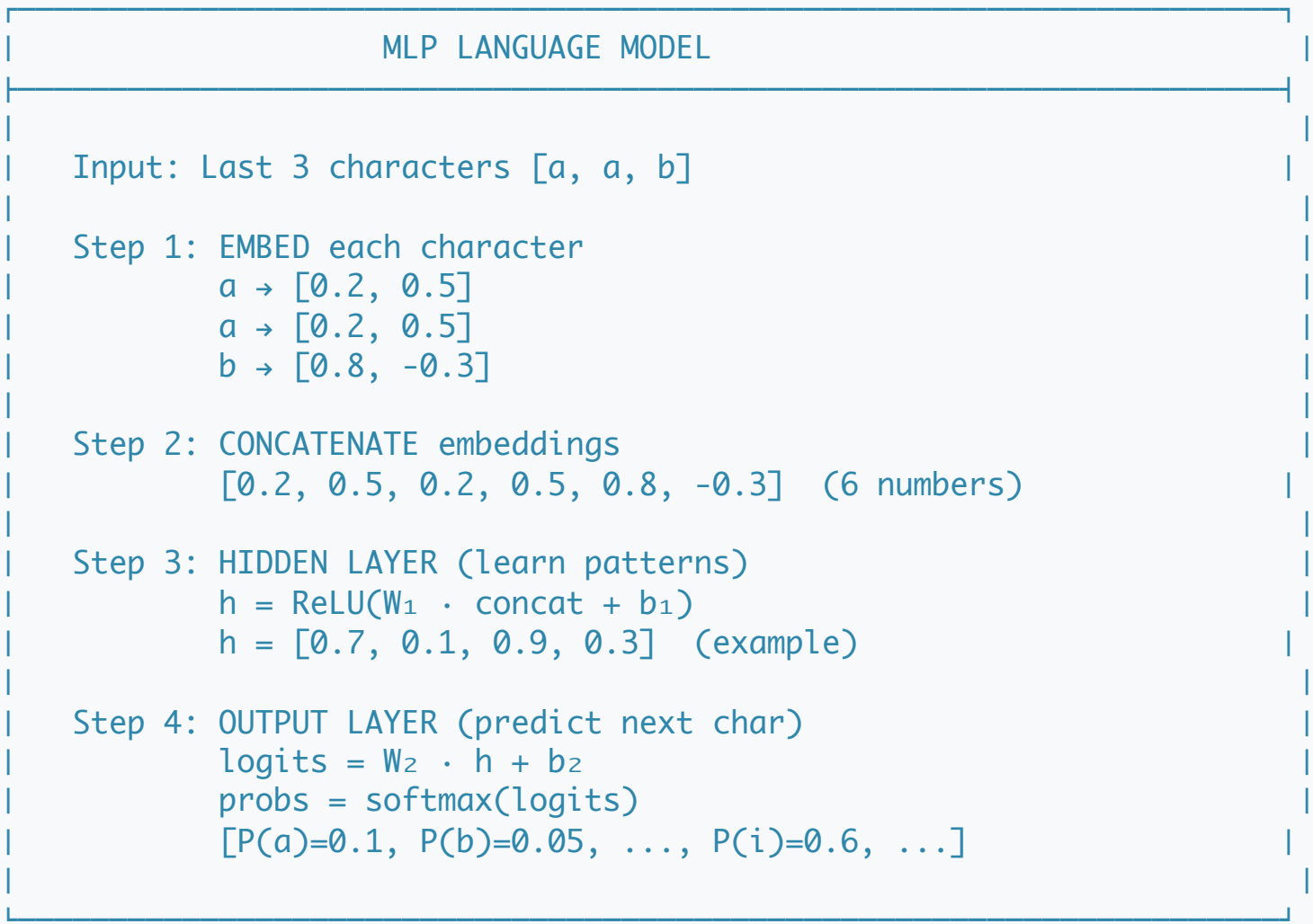
Neural Networks for Next-Token Prediction

From Counting to Learning



The Neural Network Architecture

The Architecture Unpacked



Creating Training Data: The Sliding Window

```
Text: "aabid"

Create (context → target) pairs by sliding a window:

    Position 0:  [., ., .] → 'a'     "What comes first?"
    Position 1:  [., ., a] → 'a'     "After nothing+a?"
    Position 2:  [., a, a] → 'b'     "After a, a?"
    Position 3:  [a, a, b] → 'i'     "After a, a, b?"
    Position 4:  [a, b, i] → 'd'     "After a, b, i?"
    Position 5:  [b, i, d] → '.'     "After b, i, d?"

    ┌───┬───┬───┐
    | . | a | a | ──► Context (input)
    └───┴───┴───┘
                └──► Target: 'b' (what we want to predict)
```

Training: Learning from Mistakes

```
Step 1: Forward Pass
        ┌──────────────────────────┐
        | Input: [a, a, b]         |
        | ↓                        |
        | Network predicts:        |
        | P(i)=0.10 ← Wrong!       |
        | P(z)=0.30 ← Very wrong   |
        | P(a)=0.20                |
        └──────────────────────────┘
        Actual answer: 'i'

Step 2: Compute Loss
        Loss = -log(P(correct answer))
        Loss = -log(0.10) = 2.3   ← High loss = bad prediction

Step 3: Backpropagation
        Compute gradients: "How should each weight change?"
        Adjust weights to make P(i) higher next time

Step 4: Repeat millions of times
        → Network learns to predict well!
```

The Loss Function: Cross-Entropy

```
┌──────────────────────────────────────────────────────────────────────┐
│                    CROSS-ENTROPY LOSS                                │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Loss = -log(P(correct_answer))                                     │
│                                                                      │
│   Examples:                                                          │
│                                                                      │
│   If model is confident and RIGHT:                                   │
│   P(correct) = 0.95  →  Loss = -log(0.95) = 0.05  ← Low loss!        │
│                                                                      │
│   If model is uncertain:                                             │
│   P(correct) = 0.50  →  Loss = -log(0.50) = 0.69  ← Medium loss      │
│                                                                      │
│   If model is confident and WRONG:                                   │
│   P(correct) = 0.01  →  Loss = -log(0.01) = 4.6   ← High loss!       │
│                                                                      │
│   The model gets heavily penalized for confident wrong answers!      │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

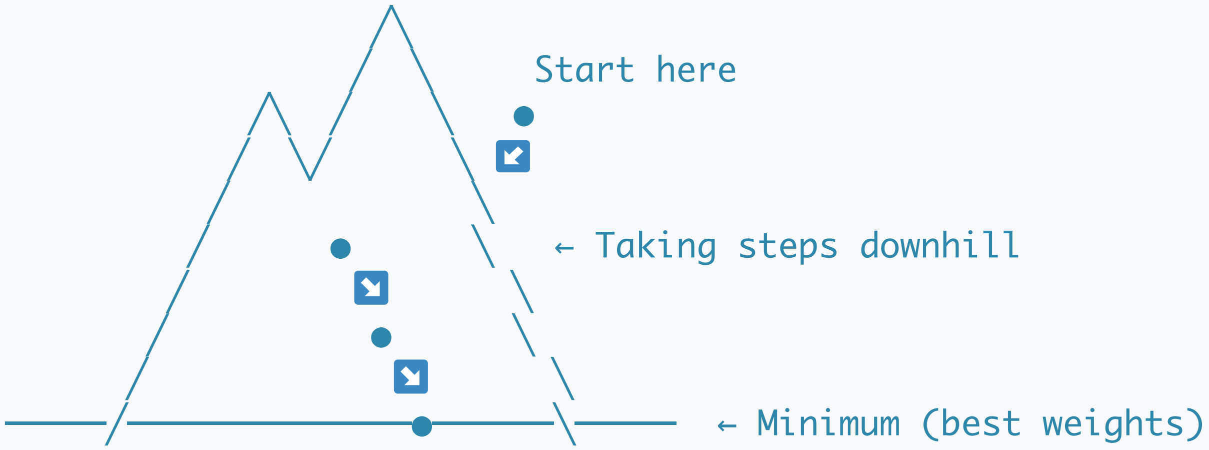
Gradient Descent: Finding the Best Weights

Gradient Descent Intuition

FINDING THE VALLEY

Imagine you're blindfolded on a mountain. Goal: reach the lowest point (minimum loss).

Strategy:
1. Feel the ground slope with your feet (compute gradient)
2. Take a step downhill (update weights)
3. Repeat until you can't go lower

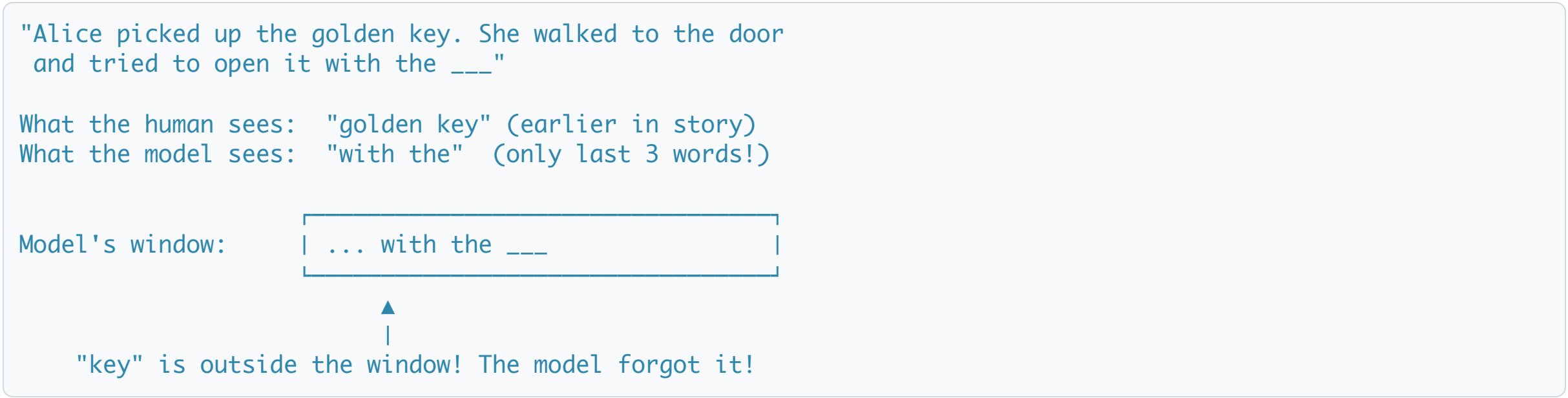


Level 5: The Context Problem

Why Fixed Windows Aren't Enough

The Fatal Flaw

Our neural network has a **fixed context window** (e.g., 3 characters).



Why Is This a Problem?

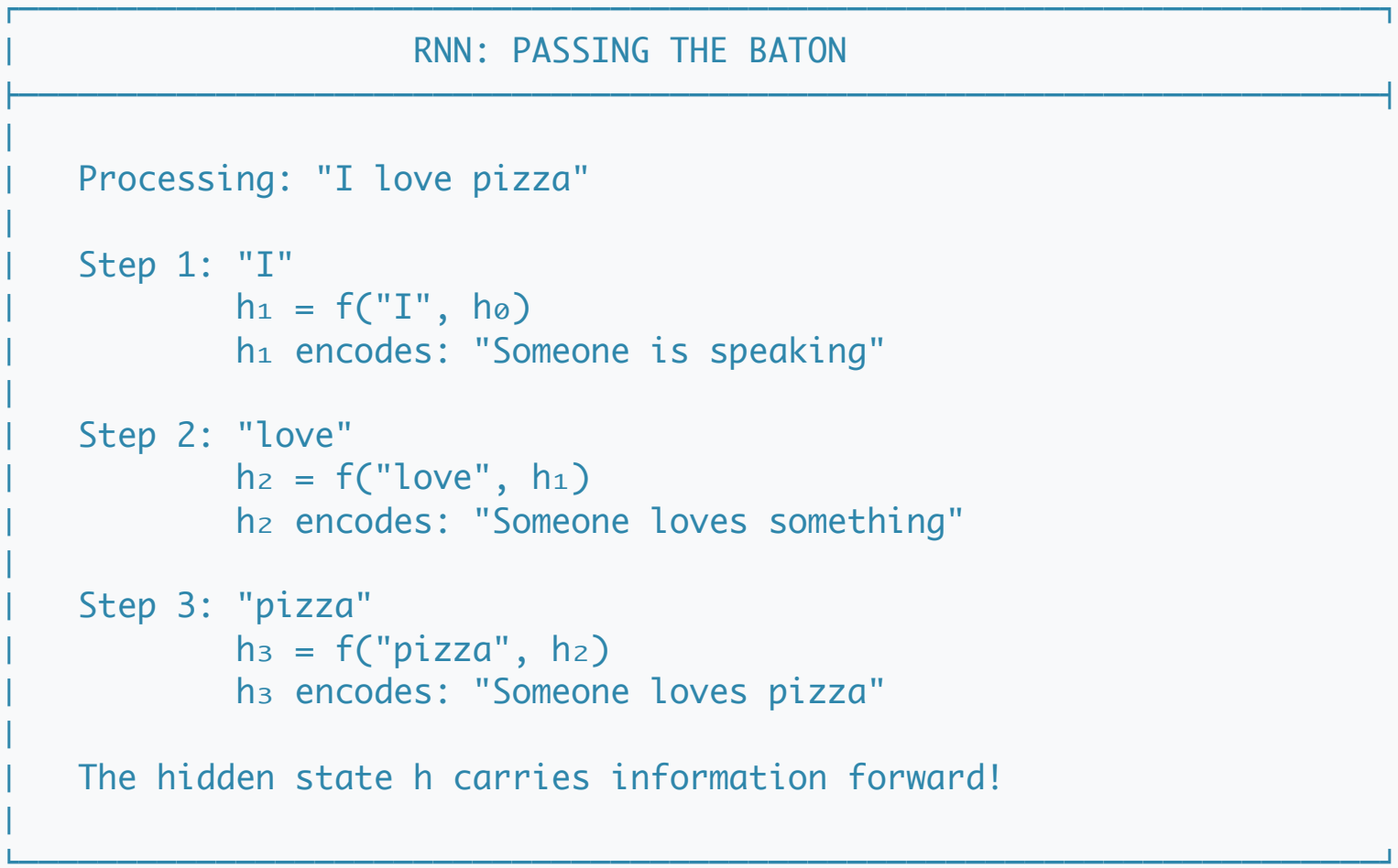
```
┌──────────────────────────────────────────────────────────────────────┐
│                      CONTEXT EXAMPLES                                │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Example 1: Pronouns                                                │
│   "John gave Mary a book. She thanked ___"                           │
│   Answer: "him" (need to remember "John")                            │
│                                                                      │
│   Example 2: Callbacks                                               │
│   "In Chapter 1, we introduced X. Now, let's explore ___ further."   │
│   Answer: "X" (need to remember earlier topic)                       │
│                                                                      │
│   Example 3: Long dependencies                                       │
│   "The cat, which was sleeping on the mat that grandmother           │
│    made last winter, suddenly ___"                                   │
│   Answer: verb about "cat" (need to skip 15 words!)                  │
│                                                                      │
│   Example 4: Instructions                                            │
│   "I want you to translate to French: Hello"                         │
│   Need to remember "translate to French" while processing "Hello"    │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Attempted Solution: RNNs

Recurrent Neural Networks maintain a "memory" that carries forward.


center

How RNNs Work

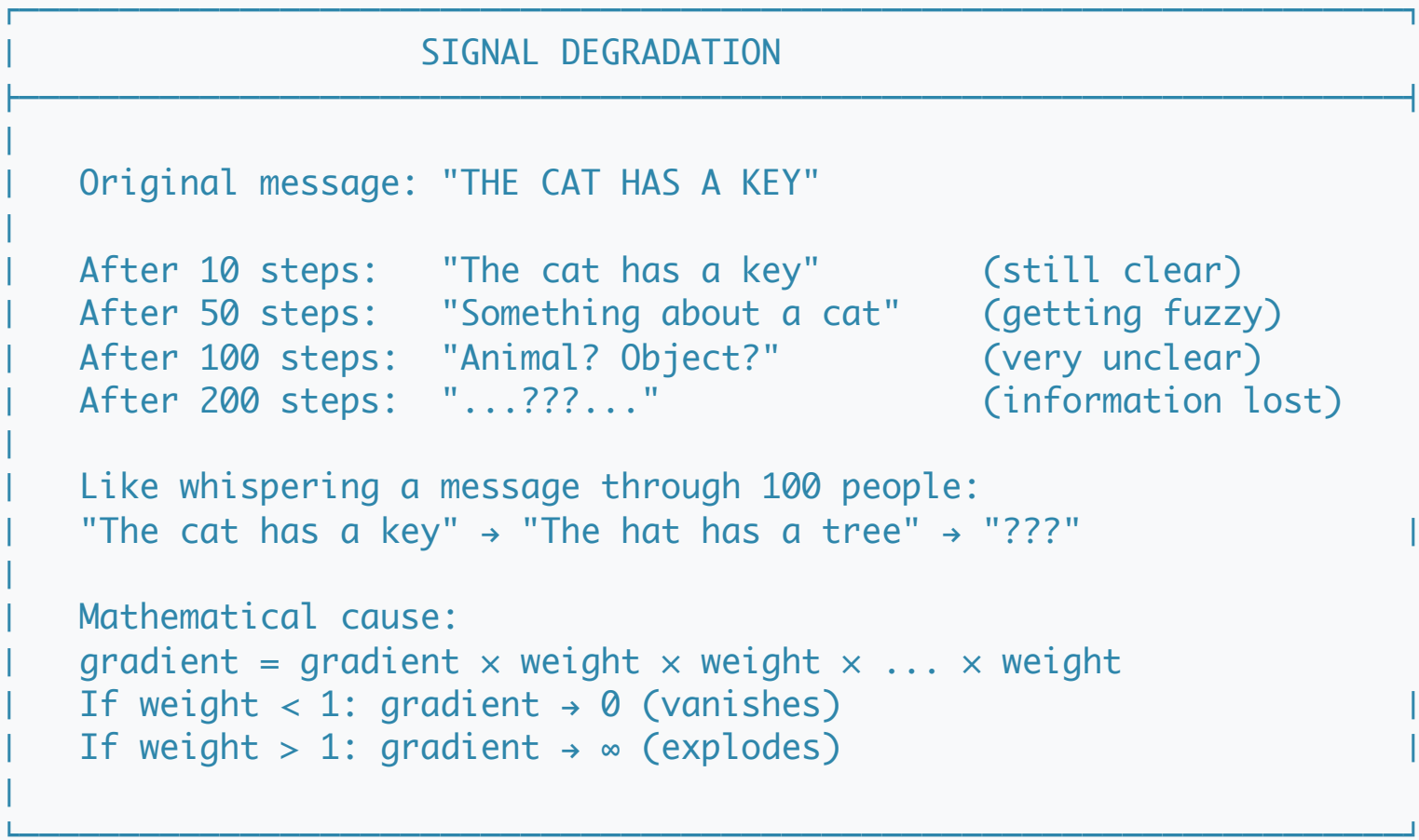


Why RNNs Still Fail

The Telephone Game Problem:

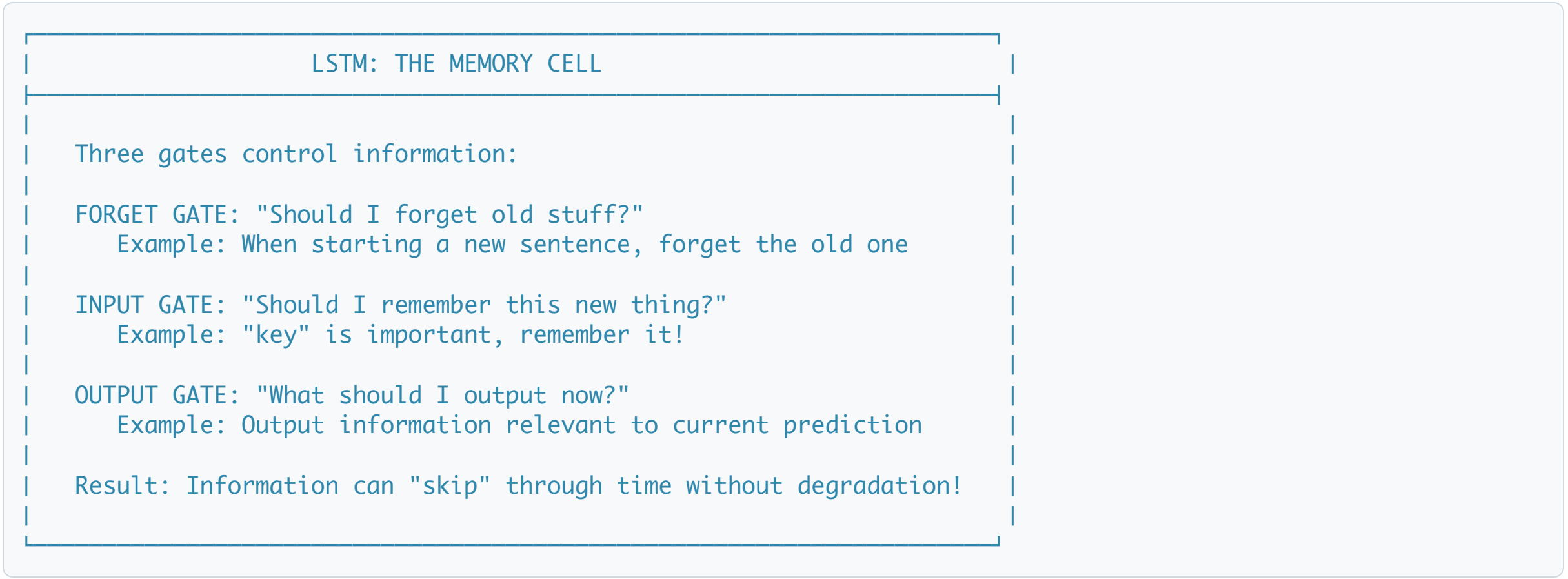
```
Start: "Alice has a key"
  |
  ▼ (pass through 50 words)
  |
  ▼ (memory gets compressed, some info lost)
  |
  ▼ (pass through 50 more words)
  |
  ▼ (even more degraded)
  |
End: "She opened the door with the ___"

By now, "key" has been corrupted or forgotten!

This is the "Vanishing Gradient" problem:
Gradients become tiny → Old info can't influence predictions
```

The Vanishing Gradient Problem



LSTM: A Better RNN

Long Short-Term Memory cells have "gates" that control information flow.



But LSTMs are still slow (sequential) and still struggle with very long contexts.

Level 6: The Revolution

Attention: "Just Look Back!"

The Brilliant Idea

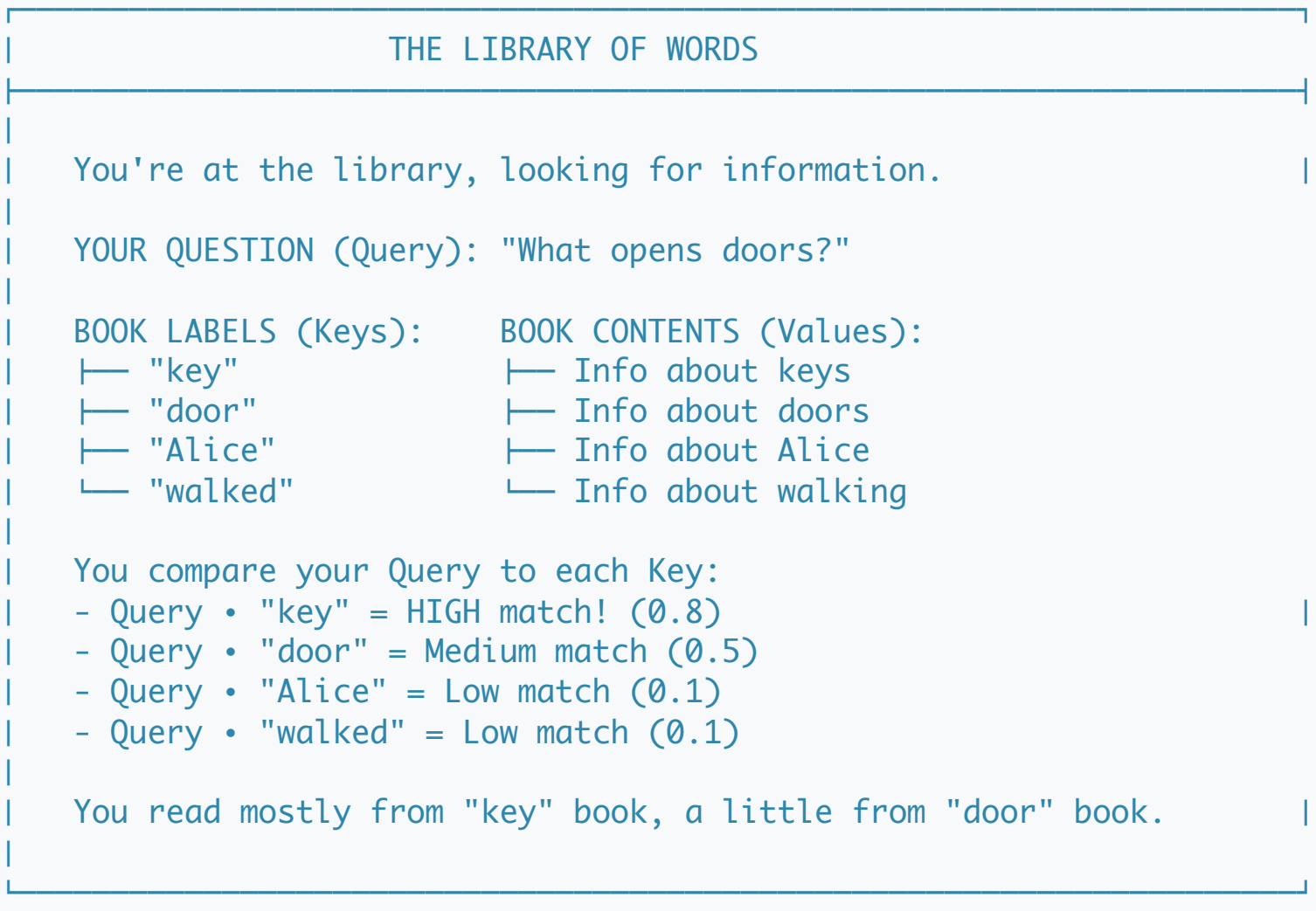
What if, instead of compressing everything into a hidden state...

We could just **look back** at everything directly?

Attention: The Searchlight Analogy

```
Reading Methods Compared:

FIXED WINDOW (MLP):
    Reading with tunnel vision
    ┌────┐
    |▒▒▒▒| ← Can only see this tiny part
    └────┘

RNN:
    Reading while trying to remember everything
    "I think there was a key... or was it a lock?"

ATTENTION:
    Reading with a highlighter and search engine!
    "Let me search for 'object that opens doors'..."
    Found: "key" at position 7!
    └── Spotlight on relevant words!
```

How Attention Works: The Library Analogy



How Attention Works: Q, K, V

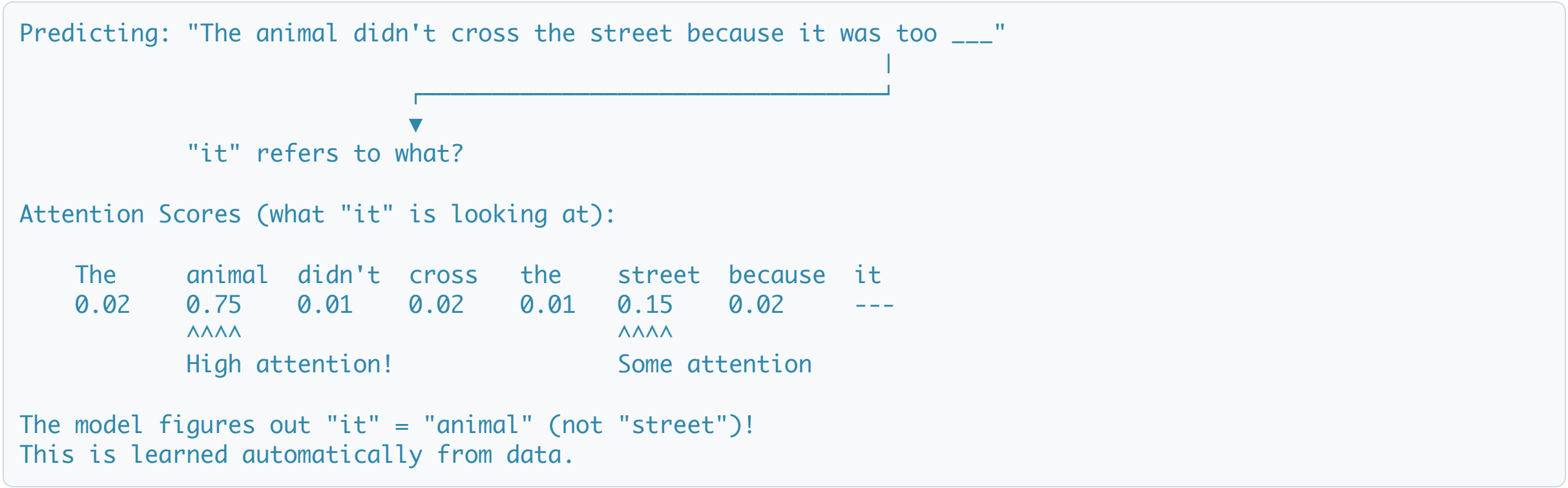
Think of it like a **database lookup**:


center

The Math of Attention

```
┌──────────────────────────────────────────────────────────────────────┐
│                      ATTENTION COMPUTATION                           │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Query (Q):  What am I looking for? (from current position)         │
│   Key (K):    What does each position offer? (from all positions)    │
│   Value (V):  What is the content? (from all positions)              │
│                                                                      │
│   Step 1: Compute similarity scores                                  │
│           scores = Q · Kᵀ                                            │
│                                                                      │
│   Step 2: Normalize with softmax                                     │
│           attention_weights = softmax(scores / √d)                   │
│                                                                      │
│   Step 3: Weighted sum of values                                     │
│           output = attention_weights · V                             │
│                                                                      │
│   Formula:  Attention(Q, K, V) = softmax(QKᵀ / √d) · V               │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Attention Scores Visualized



Why "Wide" Beats "Narrow"

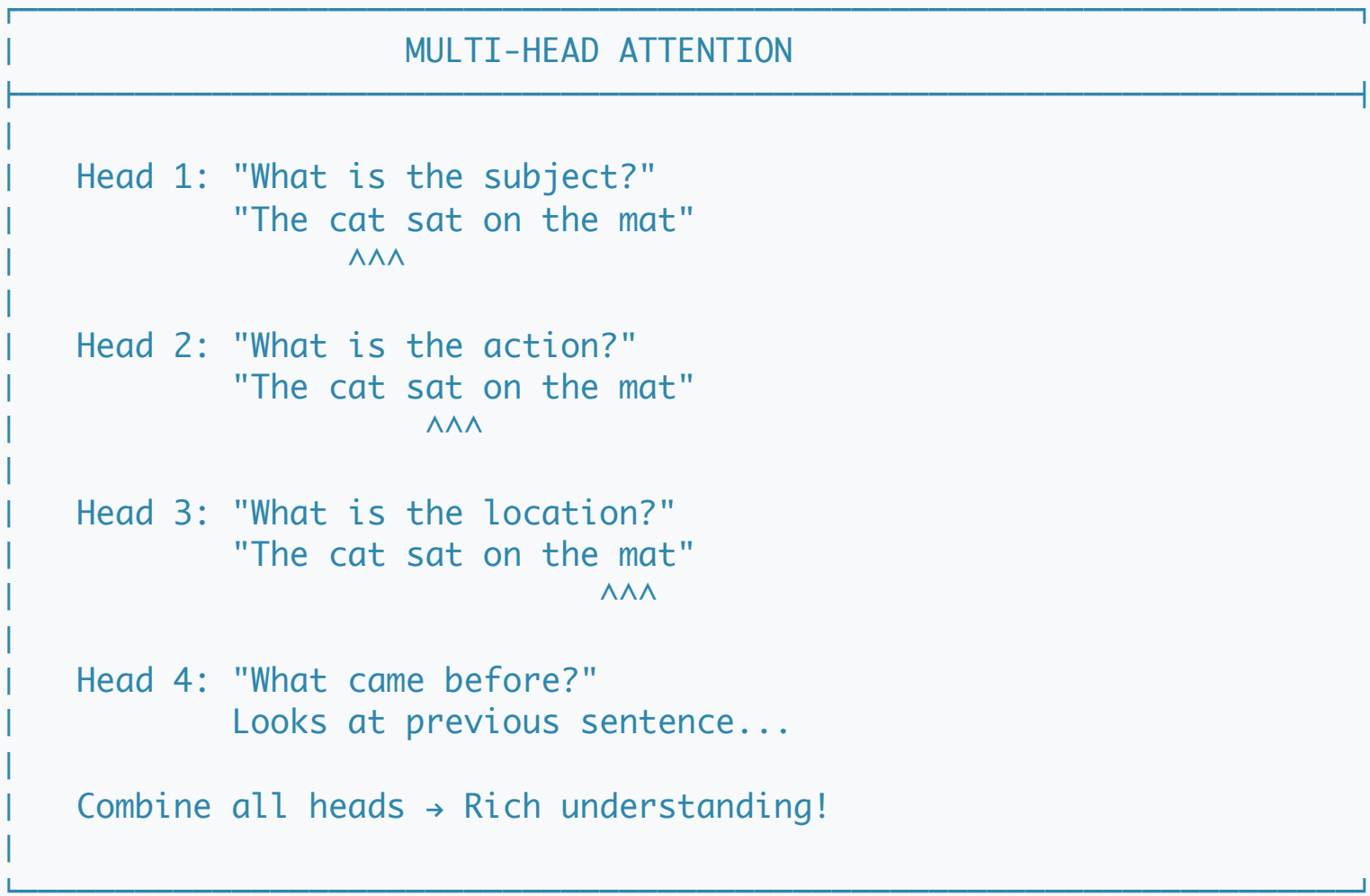
```
┌──────────────────────────────────────────────────────────────────────┐
│                        COMPARISON                                    │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Predicting the word after "The quick brown fox jumps over          │
│   the lazy dog and then runs to the ___"                             │
│                                                                      │
│   Bigram: sees only "the" → could predict anything!                  │
│                                                                      │
│   3-char window: sees "the " → still very ambiguous                  │
│                                                                      │
│   RNN: has a blurry memory of earlier words                          │
│        "something about a fox... and running?"                       │
│                                                                      │
│   Attention: can directly look at "fox", "runs", "dog"               │
│        Weights: fox(0.3), runs(0.2), lazy(0.1), ...                  │
│        "The fox is running, so it might go to a... den? forest?"     │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Self-Attention: Every Word Looks at Every Word

```
Sentence: "The cat sat on the mat"

              The   cat   sat   on   the   mat
           ┌─────┬─────┬─────┬────┬─────┬─────┐
    The    | 0.5 | 0.2 | 0.1 |0.05| 0.1 |0.05 |
    cat    | 0.1 | 0.5 | 0.2 |0.05| 0.1 |0.05 |
    sat    | 0.1 | 0.3 | 0.3 | 0.1| 0.1 | 0.1 |
    on     | 0.1 | 0.1 | 0.2 | 0.3| 0.2 | 0.1 |
    the    | 0.2 | 0.1 | 0.1 | 0.1| 0.3 | 0.2 |
    mat    | 0.1 | 0.2 | 0.1 |0.05| 0.2 | 0.35|
           └─────┴─────┴─────┴────┴─────┴─────┘

Each row: Where does this word look for context?
Each row sums to 1.0 (softmax)
Computed in parallel (not sequential like RNN)!
```

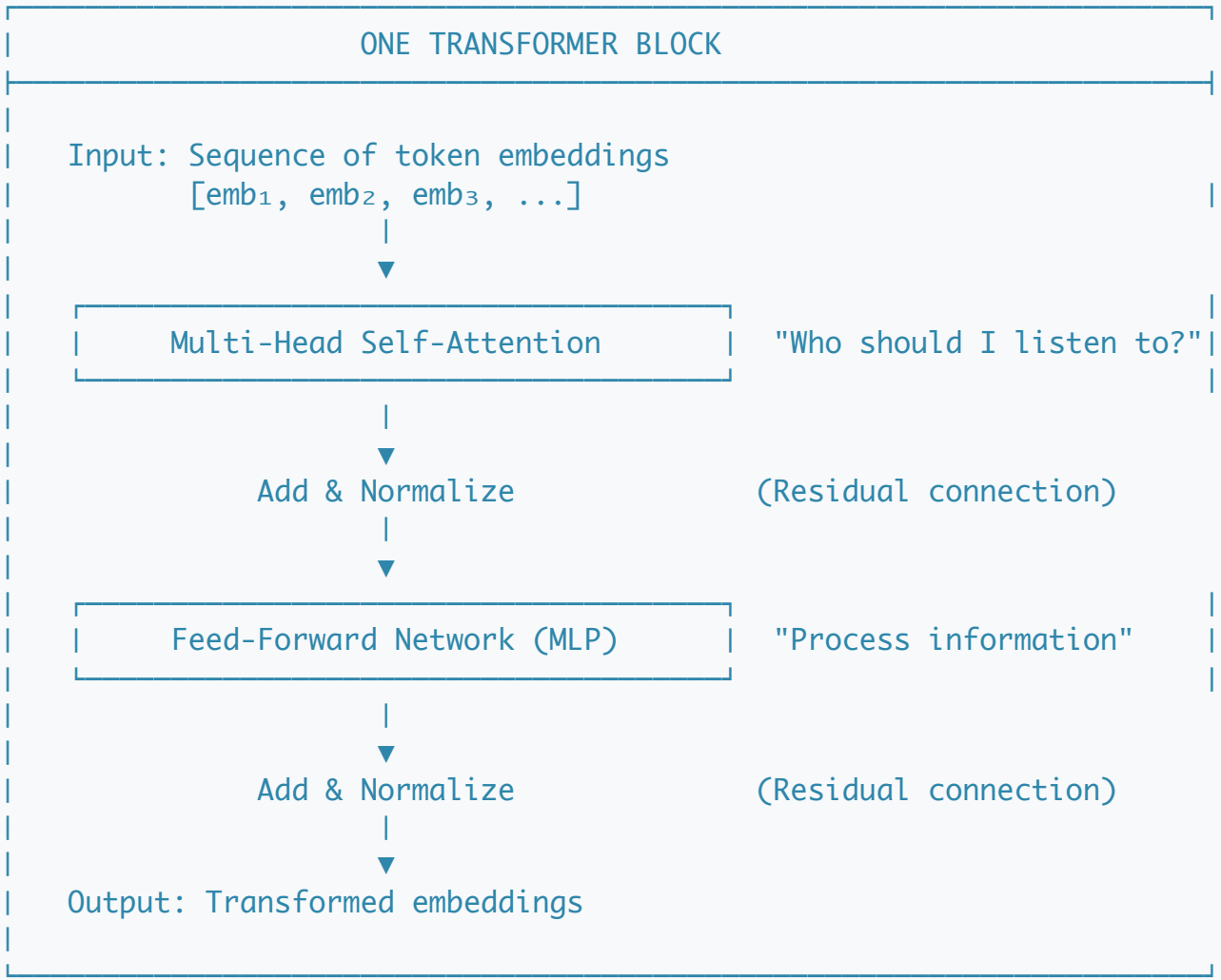
Multi-Head Attention

Idea: Different "heads" can look for different things.

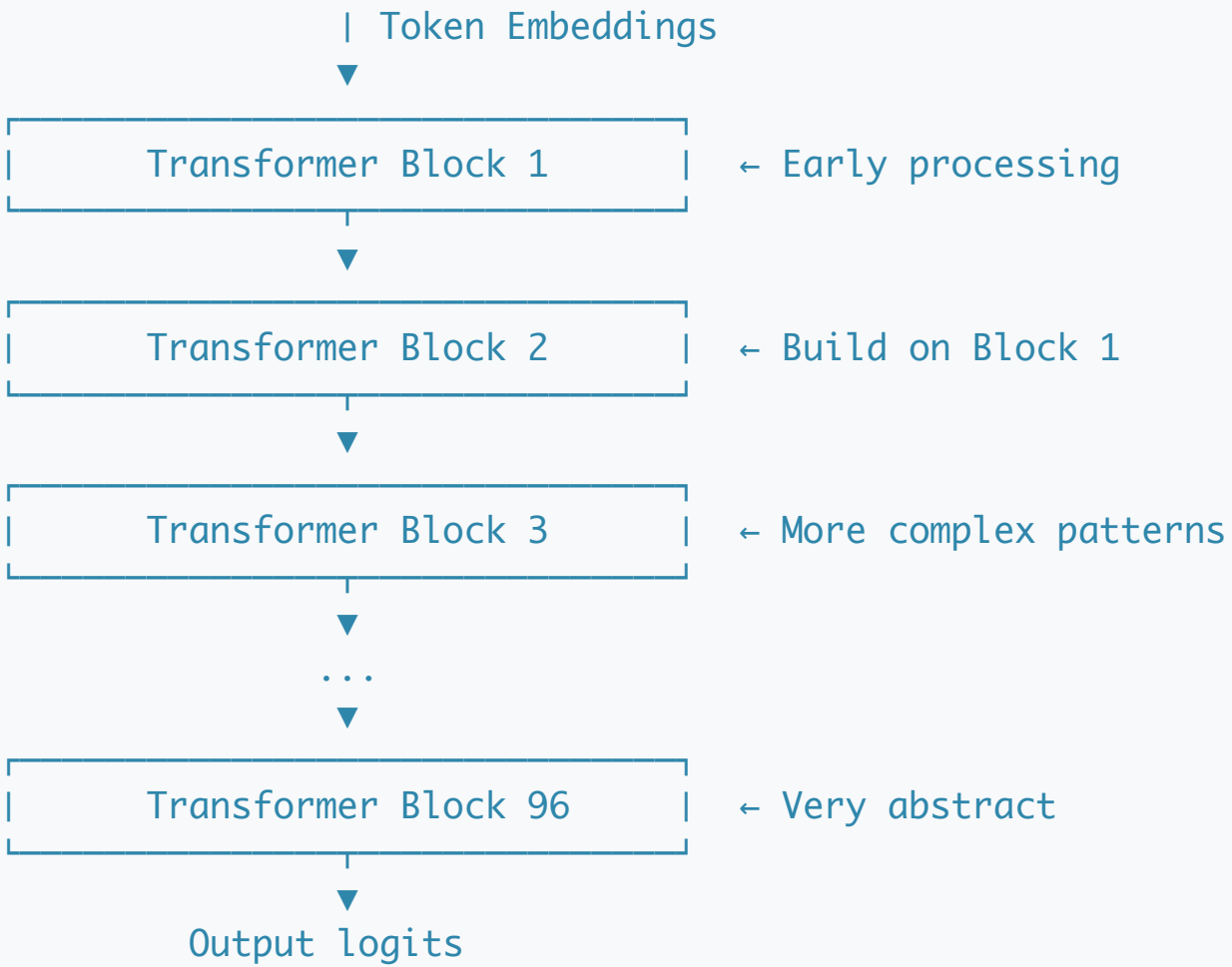


The Transformer Architecture

The Transformer Block



Stacking Transformer Blocks



GPT-4 has ~120 layers! Each layer refines understanding.

Level 7: From Theory to ChatGPT

Scaling Up

Our Toy Model vs ChatGPT

Feature	Our Toy Model	ChatGPT
Vocabulary	27 (letters)	100,000 (tokens)
Embedding Size	2 dimensions	12,288 dims
Layers	1 layer	96 layers
Attention Heads	1 head	96 heads
Parameters	~1,000	175 BILLION
Training Data	1,000 names	500B+ tokens
Context Window	3 chars	128K tokens
Training Time	1 minute	Months on 1000s of GPUs

Same core algorithm. Just **much, much bigger**.

Tokenization: Not Characters, Not Words

```
LLMs use "TOKENS" – subword units (BPE algorithm):

"unhappiness" → ["un", "happiness"]  or  ["un", "hap", "piness"]

Why?
- Characters: Too slow (many steps to generate a word)
- Words: Too many unique words (millions!)
- Tokens: Best of both worlds (~50,000-100,000 tokens)

Example:
┌──────────────────────────────────────────────────────────────┐
| Text: "ChatGPT is amazing!"                                  |
|                                                              |
| Tokens: ["Chat", "G", "PT", " is", " amazing", "!"]          |
|         [15496, 38,  2898,  318,    4998,      0]  ← Token IDs  |
└──────────────────────────────────────────────────────────────┘

Note: Spaces are often part of tokens (" is" not "is")
```

How BPE Tokenization Works

```
┌──────────────────────────────────────────────────────────────────────┐
│                    BYTE PAIR ENCODING (BPE)                          │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Start with character vocabulary: {a, b, c, ..., z, space, ...}     │
│                                                                      │
│   Algorithm:                                                         │
│   1. Count all adjacent pairs in corpus                              │
│   2. Most common pair: "th" (appears 10,000 times)                   │
│   3. Merge: create new token "th"                                    │
│   4. Repeat until vocabulary reaches target size                     │
│                                                                      │
│   After many merges:                                                 │
│   "the" → single token                                               │
│   "ing" → single token                                               │
│   "tion" → single token                                              │
│   "unhappiness" → ["un", "happiness"]                                │
│                                                                      │
│   Common words = 1 token, rare words = multiple tokens               │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Tokenization Quirks

```
┌──────────────────────────────────────────────────────────────────────┐
│                    TOKENIZATION FUN FACTS                            │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Why LLMs struggle with:                                            │
│                                                                      │
│   1. Counting letters:                                               │
│      "strawberry" → ["str", "aw", "berry"]  ← 'r' split across!      │
│      Model doesn't "see" individual letters easily                   │
│                                                                      │
│   2. Non-English languages:                                          │
│      "Hello" → 1 token                                               │
│      "नमस्ते" (Hindi) → 6 tokens  ← Same meaning, 6x tokens!         │
│                                                                      │
│   3. Numbers:                                                        │
│      "1234" might be 1 token                                         │
│      "12345" might be 2 tokens ["1234", "5"]                         │
│      Math becomes harder!                                            │
│                                                                      │
│   4. Code:                                                           │
│      "    " (4 spaces) might be 1 token                              │
│      "   " (3 spaces) might be 3 tokens                              │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Positional Encoding

How does the model know word ORDER?

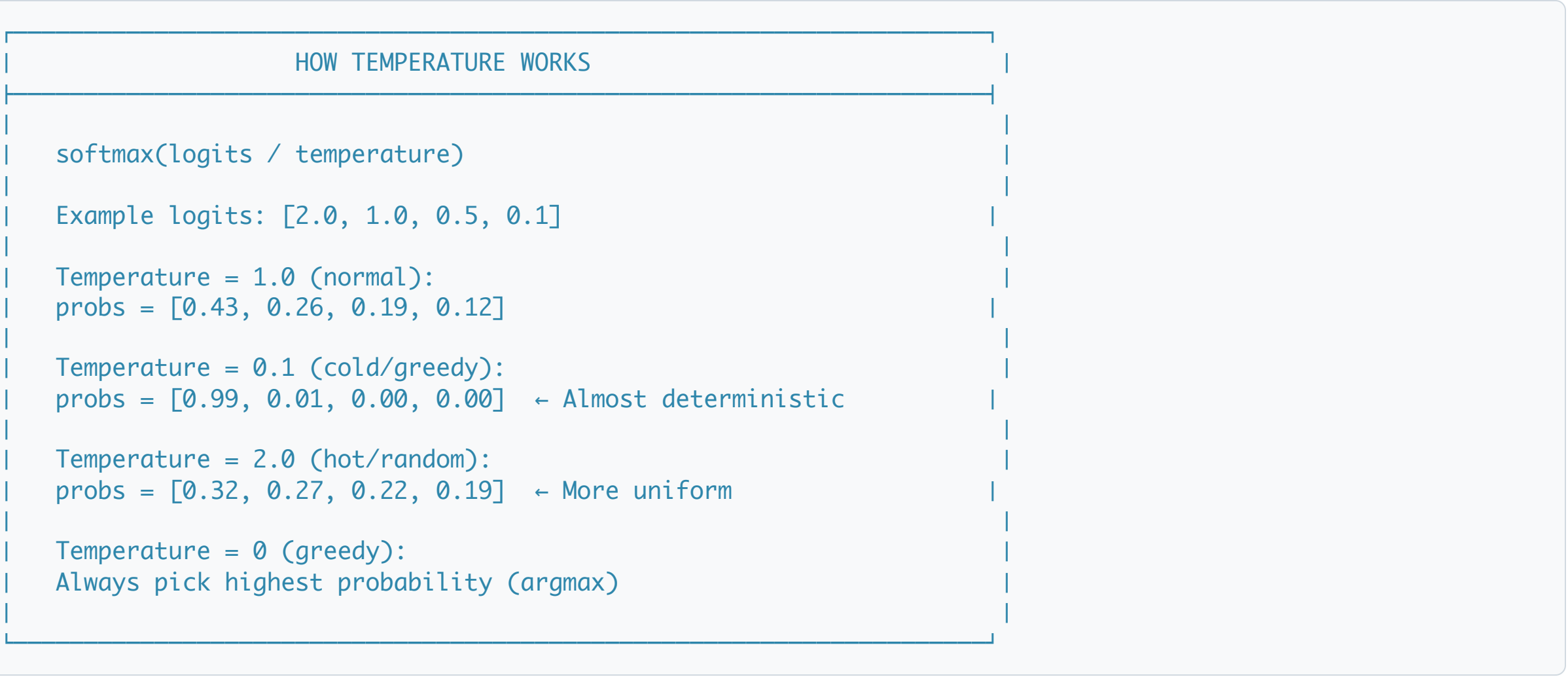
```
┌──────────────────────────────────────────────────────────────────┐
│                    POSITIONAL ENCODING                           │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│   Problem: Attention is permutation-invariant!                   │
│   "Dog bites man" and "Man bites dog" look the same to attention.│
│                                                                  │
│   Solution: Add position information to each embedding.          │
│                                                                  │
│   token_embedding("cat")       = [0.5, 0.3, 0.8, ...]            │
│   position_encoding(pos=3)     = [0.1, -0.2, 0.4, ...]           │
│   final_embedding              = [0.6, 0.1, 1.2, ...]            │
│                                                                  │
│   Now "cat" at position 3 is different from "cat" at position 10!│
│                                                                  │
│   Original paper uses sinusoidal patterns:                       │
│   PE(pos, 2i) = sin(pos / 10000^(2i/d))                          │
│   PE(pos, 2i+1) = cos(pos / 10000^(2i/d))                        │
│                                                                  │
│   Modern models: Learn position embeddings!                      │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Temperature: The Creativity Knob

When sampling the next token, we can adjust **temperature**:

```
Probabilities for next word after "I love to eat":

                Low Temp (0.1)      High Temp (2.0)
                (Conservative)      (Creative)
┌──────────────┬──────────────────┬──────────────────┐
|   pizza      |     0.80         |     0.25         |
|   pasta      |     0.15         |     0.20         |
|   shoes      |     0.01         |     0.15         |
|   clouds     |     0.001        |     0.12         |
|   dreams     |     0.0001       |     0.10         |
└──────────────┴──────────────────┴──────────────────┘

Low temp → Always picks "pizza" (boring but safe)
High temp → Might pick "clouds" (creative but weird)
```

Temperature: The Math



Top-k and Top-p Sampling

```
┌──────────────────────────────────────────────────────────────────────┐
│                    OTHER SAMPLING METHODS                            │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   TOP-K SAMPLING:                                                    │
│   Only consider the top K most likely tokens.                        │
│                                                                      │
│   All tokens: pizza(0.4), pasta(0.3), shoes(0.1), clouds(0.05)...    │
│   Top-3:      pizza(0.53), pasta(0.40), shoes(0.07)                  │
│                                                                      │
│   Problem: K is fixed. Sometimes 3 options make sense, sometimes 10. │
│                                                                      │
│   TOP-P (NUCLEUS) SAMPLING:                                          │
│   Include tokens until cumulative probability > P.                   │
│                                                                      │
│   P = 0.9:                                                           │
│   Include pizza(0.4) + pasta(0.3) + shoes(0.1) = 0.8 < 0.9           │
│   Include clouds(0.05) → 0.85 < 0.9                                  │
│   Include dreams(0.04) → 0.89 < 0.9                                  │
│   Include hope(0.02) → 0.91 > 0.9 → Stop!                            │
│                                                                      │
│   Adaptive: Narrow when confident, wide when uncertain!              │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

The Sampling Tree

Because we sample probabilistically, each generation is different!


center

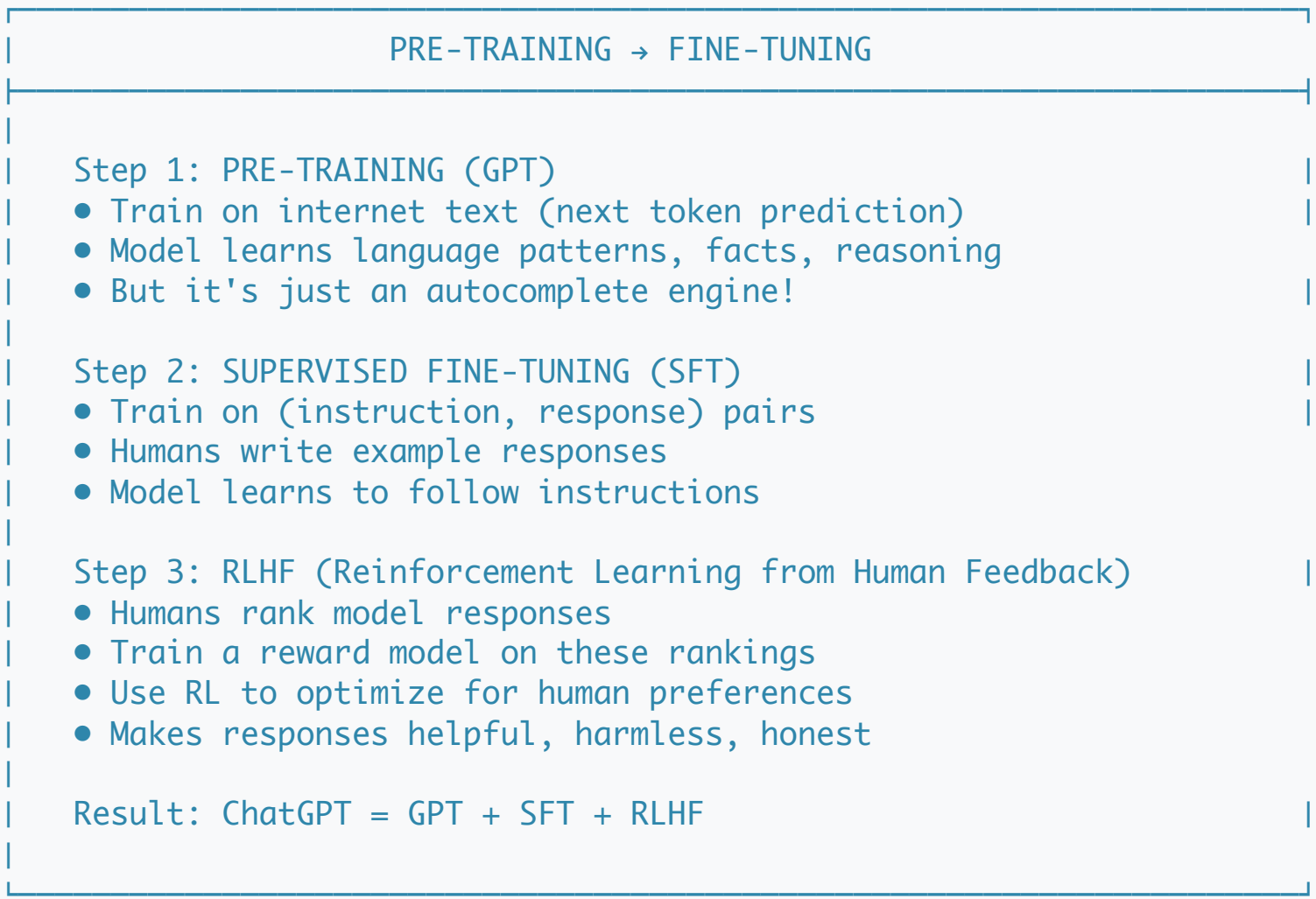
Training at Scale

```
┌──────────────────────────────────────────────────────────────────────┐
│                        GPT-3 TRAINING                                │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   Data:                                                              │
│   • Common Crawl (web pages): 410B tokens                            │
│   • Books: 67B tokens                                                │
│   • Wikipedia: 3B tokens                                             │
│   • Total: ~500B tokens                                              │
│                                                                      │
│   Compute:                                                           │
│   • 10,000 GPUs                                                      │
│   • Training time: ~1 month                                          │
│   • Cost: ~$4.6 million (just electricity!)                          │
│                                                                      │
│   Model:                                                             │
│   • 175 billion parameters                                           │
│   • 96 layers                                                        │
│   • 96 attention heads                                               │
│   • 12,288 embedding dimensions                                      │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

The Complete Recipe


center

From GPT to ChatGPT



Summary: The Full Stack

```
Layer 0: THE TASK
         Predict P(next_token | all_previous_tokens)

Layer 1: REPRESENTATION
         Tokens → Embeddings (meaning as vectors)

Layer 2: CONTEXT
         Self-Attention (look at relevant past tokens)

Layer 3: COMPUTATION
         Feed-Forward layers (process information)

Layer 4: STACKING
         Repeat attention+FFN 96 times for depth

Layer 5: TRAINING
         Next token prediction on internet-scale data

Layer 6: ALIGNMENT
         Instruction tuning + RLHF for helpfulness
```

Key Takeaways

```
┌──────────────────────────────────────────────────────────────────────┐
│                      THE 5 BIG IDEAS                                 │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│   1. PREDICTION IS ALL YOU NEED                                      │
│      Just predicting the next token gives emergent abilities         │
│                                                                      │
│   2. EMBEDDINGS CAPTURE MEANING                                      │
│      Similar words → Similar vectors                                 │
│                                                                      │
│   3. ATTENTION ENABLES LONG-RANGE CONTEXT                            │
│      Every token can look at every other token                       │
│                                                                      │
│   4. SCALE MATTERS                                                   │
│      Bigger models + more data = better capabilities                 │
│                                                                      │
│   5. ALIGNMENT IS CRUCIAL                                            │
│      Raw prediction → helpful assistant through RLHF                │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Resources to Learn More

Videos:

1. **Andrej Karpathy** - "Neural Networks: Zero to Hero"
    - Builds GPT from scratch
2. **3Blue1Brown** - "Attention in Transformers"
    - Beautiful animations

Code & Blogs:

1. **NanoGPT** - Karpathy's GitHub
    - Full GPT in ~300 lines
2. **Jay Alammar** - "The Illustrated Transformer"
    - Best visualizations
3. **HuggingFace Course**
    - Practical transformer tutorials

What's Next?

In the Labs:

- Lab 4: Build bigram & neural LM
- Lab 5: Deploy with Gradio
- Generate names, explore temperature

Beyond:

- Fine-tune a real LLM
- Build RAG applications
- Explore multimodal models

> The same simple idea — predicting the next token — powers everything from autocomplete to ChatGPT to Claude. Now you understand how!

Thank You!

"The best way to predict the future is to create it."

The same simple idea — predicting the next token — powers everything from autocomplete to ChatGPT to Claude.

Questions?