Next Token Prediction

Building ChatGPT from Scratch (Conceptually)

Nipun Batra | IIT Gandhinagar

What We'll Learn Today

The Journey to GPT

Level	Topic	Key Concept
1	The Intuition	What does "predicting the next word" really mean?
2	The Counting Era	Bigrams: Count letter pairs
3	Representing Meaning	Embeddings: Words as vectors in space
4	Learning Patterns	Neural networks for next-token prediction
5	The Context Problem	Why we need to remember more
6	The Revolution	Attention and Transformers
7	From Theory to ChatGPT	Scaling up to billions of parameters

Level 1: The Intuition

What Are We Really Doing?

The Core Problem

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

> **"Given what I have seen so far, what word comes next?"**

Example: "The capital of France is ___" → **"Paris"**

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

You Already Know This!

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

Application	You type...	Suggestions
Phone Keyboard	"I'm running ___"	`[late]` `[out]` `[away]`
Google Search	"how to make ___"	money, pancakes, friends
Gmail Smart Compose	"Thanks for the ___"	*quick response!*

> All of these are next-word prediction models!

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:

Word	$P(\text{word} \mid \text{context})$	Confidence
Paris	0.85	
the	0.02	
London	0.01	
beautiful	0.01	
...	0.11	

All probabilities sum to 1.0

> 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	Prediction	Domain
"F = m"	"a"	Newton's Law
"To be or not to"	"be"	Shakespeare
"E = mc"	"²"	Einstein
"print('Hello"	"')"	Python syntax
"2 + 2 ="	"4"	Math
"The mitochondria is"	"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 One Algorithm

```python
def generate_text(prompt, model):
    tokens = tokenize(prompt)

    while not done:
        # Step 1: Model predicts probabilities for ALL possible next tokens
        probs = model(tokens)      # e.g., {"the": 0.3, "a": 0.2, "cat": 0.1, ...}

        # Step 2: Sample one token based on probabilities
        next_token = sample(probs)  # e.g., "the"

        # Step 3: Add to our sequence and repeat
        tokens.append(next_token)

    return tokens
```

That's ALL ChatGPT Does!

Step	What Happens	Example
Input	User types prompt	"The capital of France is"
Loop 1	Model predicts → samples	→ "Paris"
Loop 2	Model predicts → samples	→ "."
Loop 3	Model predicts → samples	→ "It"
Loop 4	Model predicts → samples	→ "is"
...	Keep going...	"known for the Eiffel Tower."

ChatGPT = this loop running billions of times with a REALLY good probability model

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** in training data!

> 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**:

Model Size	Emergent Capabilities
Small (100M params)	Complete simple sentences, basic grammar
Medium (1B params)	Answer factual questions, simple reasoning
Large (100B+ params)	Complex reasoning, code generation, creative writing, multi-step problem solving, "understanding" context

> 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

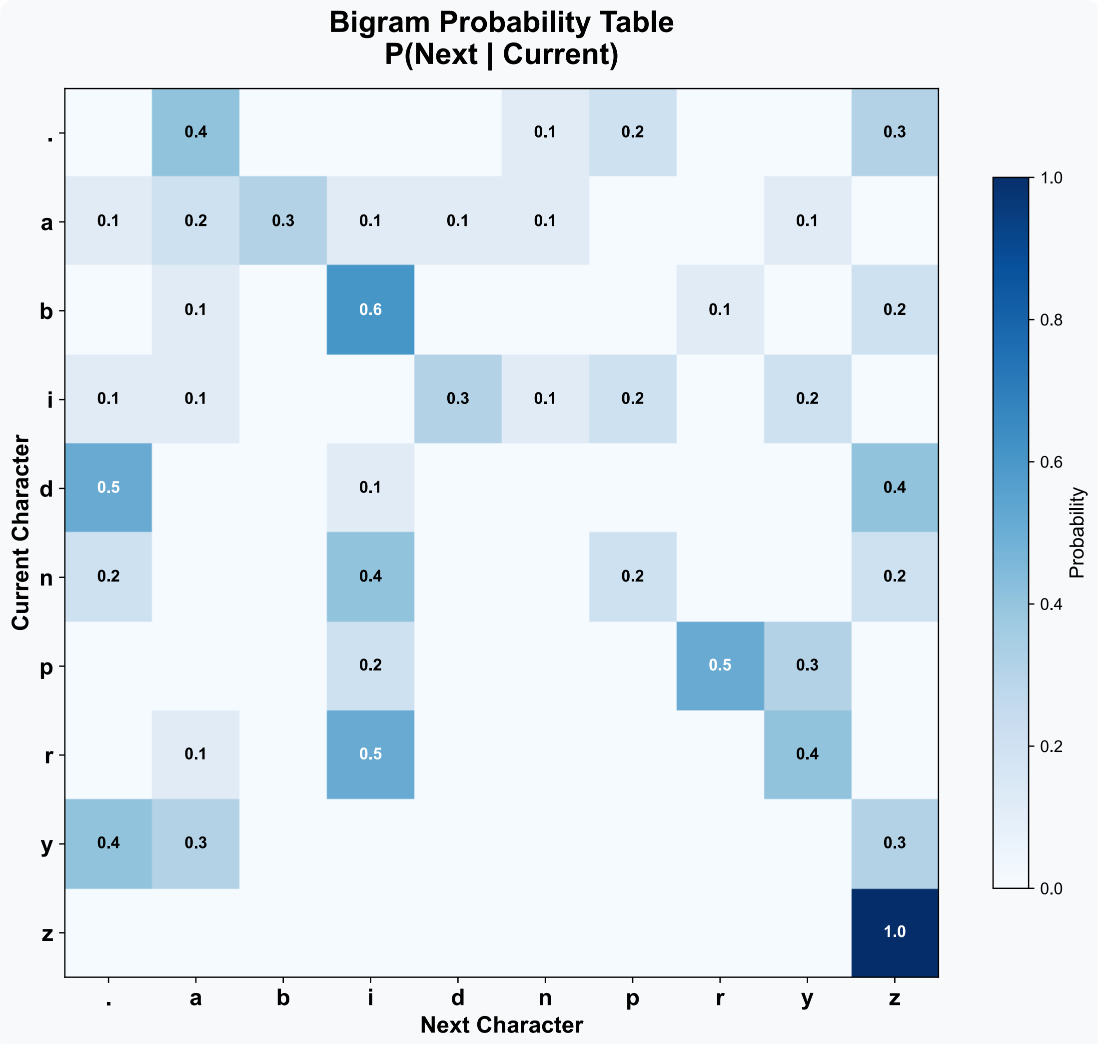
```
Training Data: "aabid", "priya", "zeel", "nipun"

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



Each row sums to 1.0 — **To generate:** Look up current letter → Sample from that row

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

Step	Current	Top Options	Roll	Selected
1	`.`	a(0.25), m(0.20), s(0.15)	0.18	m
2	`m`	a(0.40), i(0.25), o(0.15)	0.32	a
3	`a`	n(0.20), r(0.18), l(0.15)	0.45	r
4	`r`	i(0.25), a(0.20), y(0.18)	0.52	y
5	`y`	.(0.45), a(0.15), i(0.10)	0.30	.

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?

Model	What it sees	Problem
Bigram	Just "h" → ?	Doesn't even know it's in "with"!
Smarter model	Full context	Can reason about toppings vs companions

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?

N-gram	Entries (27 chars)	Feasibility
1-gram	27	Fits in memory
2-gram	$27^2 = 729$	Still fine
3-gram	$27^3 = 19{,}683$	OK
4-gram	$27^4 = 531{,}441$	Getting big
5-gram	$27^5 = 14{,}348{,}907$	Very big
10-gram	$27^{10} \approx$ 205 TRILLION	Impossible!

For words (50,000 vocabulary): 2-gram = 2.5B, 3-gram = 125T entries!

> 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: $\text{Distance}(a, b) = \text{Distance}(a, z) = \sqrt{2}$

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**.

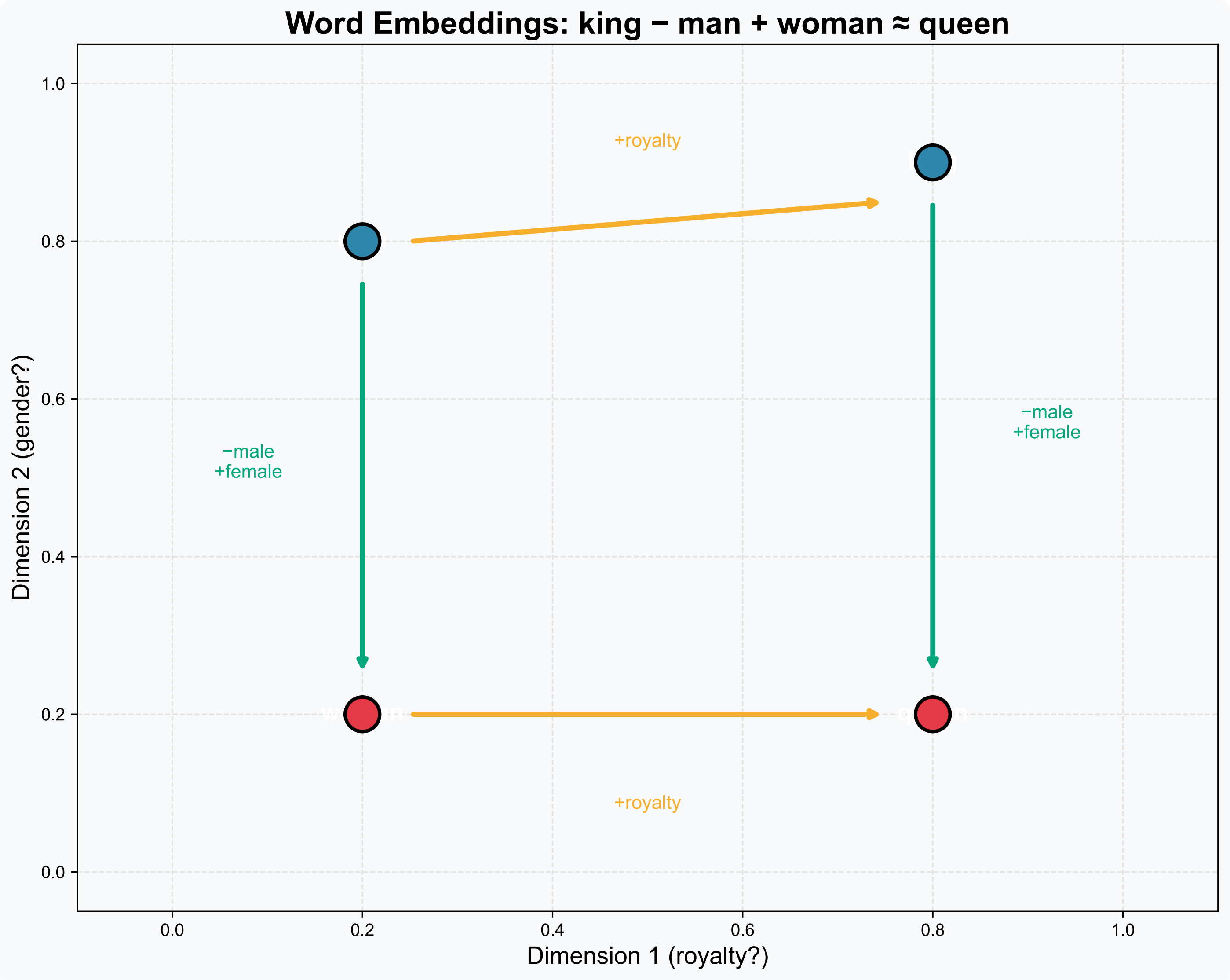
Dense Embeddings: Meaning as Coordinates

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

```
          ▲ Dimension 2 ("Common ending?")
          │
  0.9     │    ● e        ● a
          │        ● i                Vowels cluster
  0.5     │              ● o            together!
          │                    ● u
  0.1     │
          │  ● b    ● c   ● d    ● f    ← Consonants cluster
 -0.5     │            ● x    ● z       ← Rare letters
          │
          └─────────────────────────────► Dimension 1 ("Vowel?")
               -1       0       1
```

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

Word Embeddings: The Famous Example



The King - Man + Woman = Queen Example

Word Arithmetic

Word	Vector	Meaning
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

Calculation: $\text{king} - \text{man} + \text{woman} = ?$

$$[0.8, 0.3, 0.9] - [0.1, 0.3, 0.5] + [0.1, 0.9, 0.5] = [0.8, 0.9, 0.9]$$

Nearest word to $[0.8, 0.9, 0.9]$: **"queen"!**

> The model learned: "The relationship between king and man is the same as the relationship between queen and woman"

More Word Analogies

Analogy	Answer
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

Start: Random vectors for each word

Training on: "The cat sat on the mat"

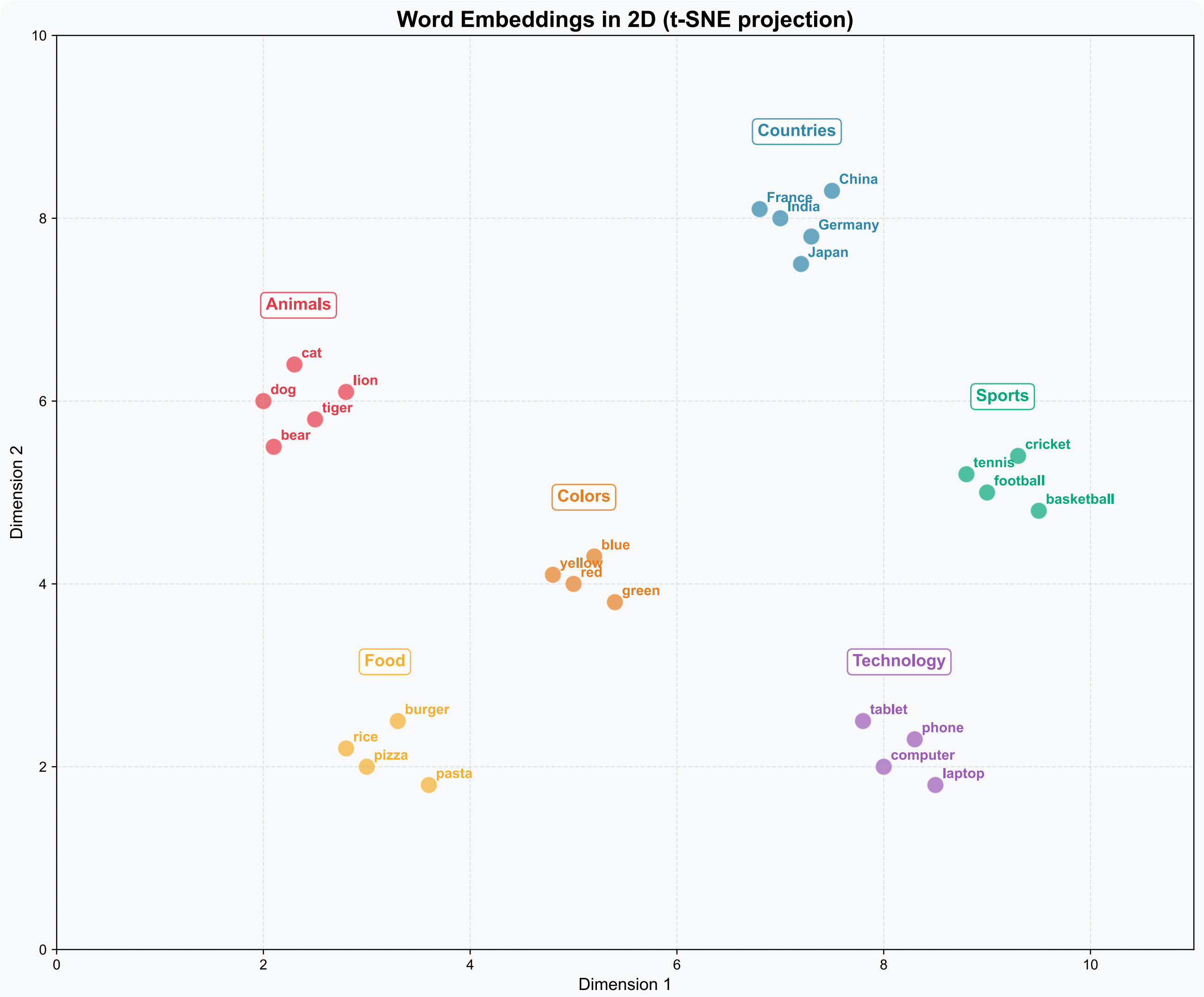
Observation	Action
"cat" often appears near "sat", "dog", "pet"	Push embeddings **closer**
"cat" rarely appears near "quantum", "fiscal"	Push embeddings **apart**

After billions of examples:

- Similar words → Similar vectors
- Related concepts → Close in space

Visualizing Embeddings

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



Embedding Dimensions

What Do Dimensions Mean?

We use 256-4096 dimensions in practice, but imagine 4:

Dim	Meaning
1	Is it alive?
2	Is it a person?
3	Concrete vs abstract?
4	Positive/negative sentiment?

Word	Vector	Interpretation
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

In reality, dimensions are learned and not so interpretable!

Level 4: Learning Patterns

Neural Networks for Next-Token Prediction

From Counting to Learning

BIGRAM (Counting)

$$P(b|a) = \frac{\text{Count}(b|a)}{\text{Count}(a)}$$

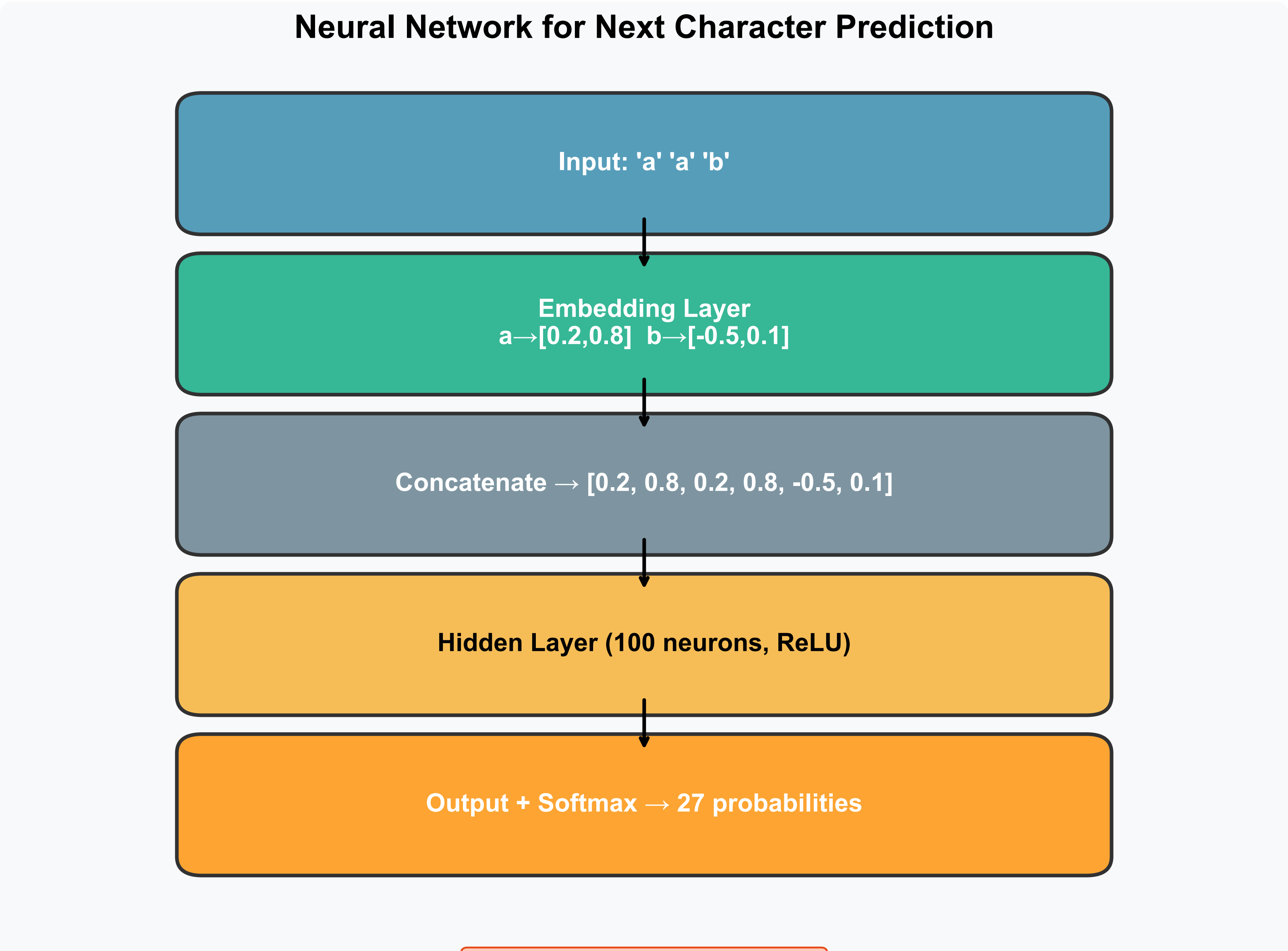
- Fixed lookup table
- Only memorizes exact patterns

NEURAL NETWORK (Learning)

$$P(\text{next}|a) = f(\text{embed}(a); \theta)$$

- Learned weights θ
- Can **generalize** to unseen patterns!

The Neural Network Architecture



The Architecture Unpacked

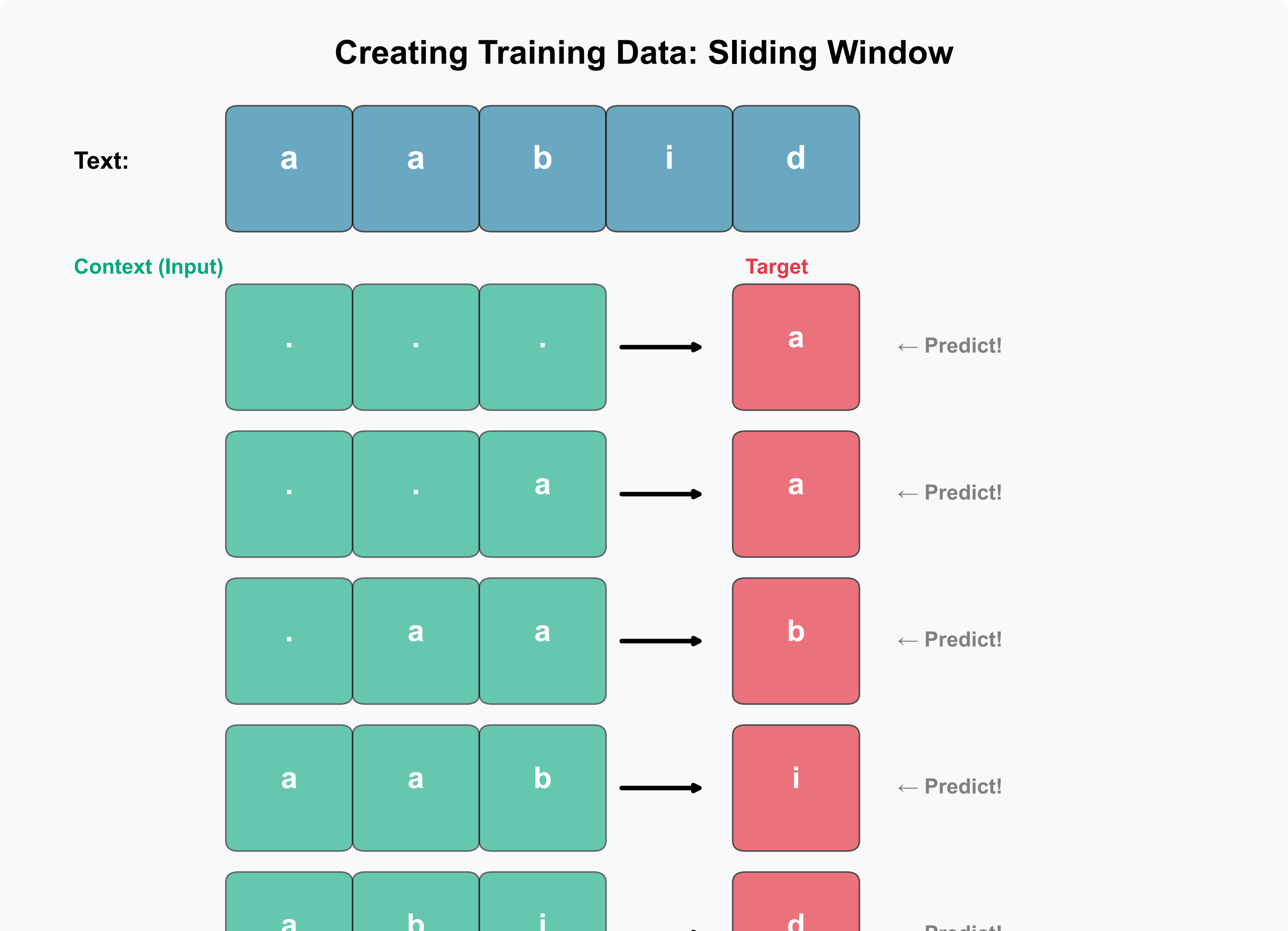
MLP Language Model — Input: Last 3 characters `[a, a, b]`

Step	Operation	Example
1. Embed	Look up each character	a → [0.2, 0.5], b → [0.8, -0.3]
2. Concatenate	Join all embeddings	[0.2, 0.5, 0.2, 0.5, 0.8, -0.3]
3. Hidden Layer	$\mathbf{h} = \mathrm{ReLU}(W_1 \cdot \mathrm{concat} + b_1)$	[0.7, 0.1, 0.9, 0.3]
4. Output Layer	$\mathrm{logits} = W_2 \cdot \mathbf{h} + b_2$	P(a)=0.1, P(b)=0.05, ..., P(i)=0.6

> The softmax converts logits to probabilities that sum to 1.

Creating Training Data: The Sliding Window

Text: `"aabid"` — Create (context → target) pairs by sliding a window:



Training: Learning from Mistakes

Step	Action	Example
1. Forward Pass	Input: [a, a, b] → Network predicts	P(i)=0.10, P(z)=0.30, P(a)=0.20
2. Compute Loss	$\text{Loss} = -\log(P(\text{correct}))$	$-\log(0.10) = 2.3$ (high = bad!)
3. Backpropagation	Compute gradients for all weights	"How should each weight change?"
4. Update Weights	Adjust to make P(i) higher	Repeat millions of times!

$**$Actual answer:** 'i' — The model was only 10% confident, so it gets a high loss and learns to do better!

The Loss Function: Cross-Entropy

$$\text{Loss} = -\log(P(\text{correct answer}))$$

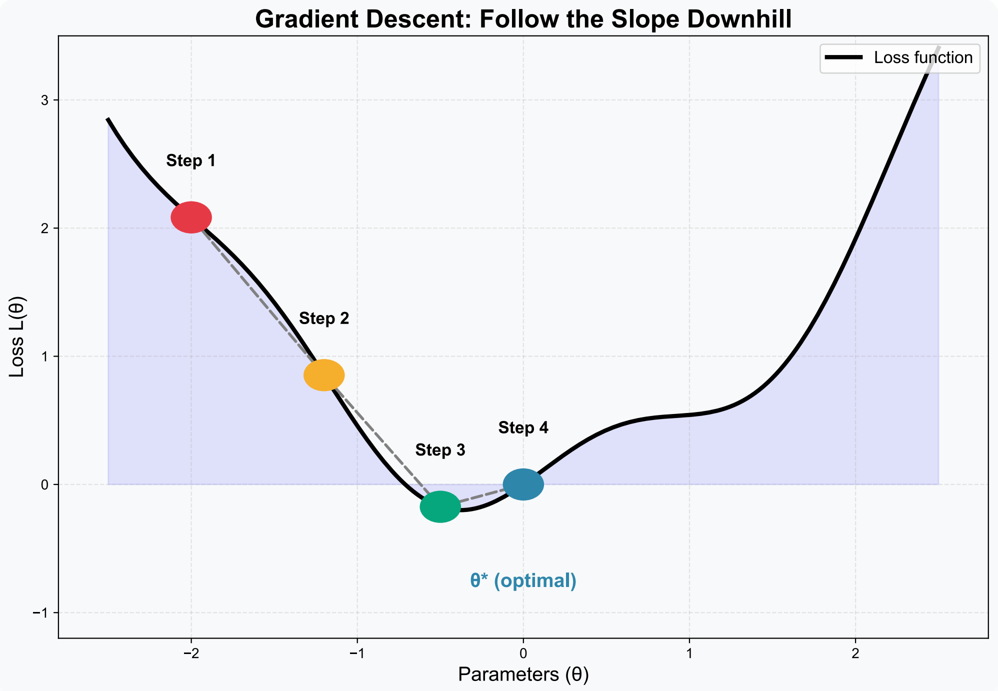
Scenario	P(correct)	Loss	Interpretation
Confident and **RIGHT**	0.95	$-\log(0.95) = 0.05$	Low loss ✓
Uncertain	0.50	$-\log(0.50) = 0.69$	Medium loss
Confident and **WRONG**	0.01	$-\log(0.01) = 4.6$	High loss ✗

The model gets heavily penalized for confident wrong answers! This encourages well-calibrated uncertainty.

Gradient Descent: Finding the Best Weights

Analogy: You're blindfolded on a mountain. Goal: reach the lowest point.

Step	Action
1	Feel slope → ∇L
2	Step downhill → $\theta - \alpha\nabla L$
3	Repeat until minimum



Learning rate (α) controls step size: too big = overshoot, too small = slow.

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).

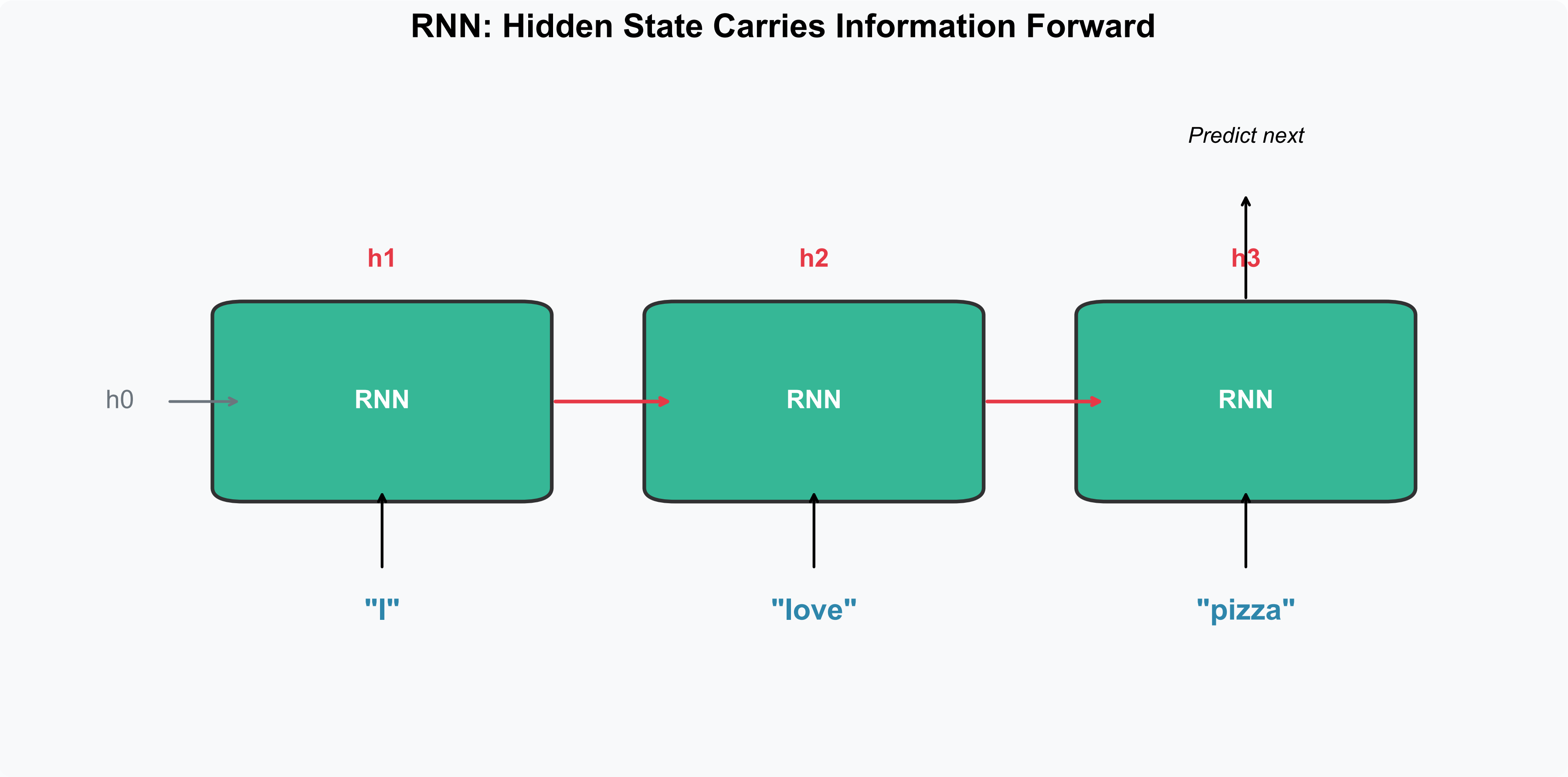
Story: "Alice picked up the golden key. She walked to the door and tried to open it with the ___"

Viewer	What They See	Problem
Human	"golden key" (earlier in story)	Full context
Model	"with the" (only last 3 words!)	**"key" is outside the window!**

> The model forgot the key! Fixed windows lose important information from earlier context.

The Solution: RNNs (The Relay Race)

Idea: Pass information forward like a **baton in a relay race**.



RNN Intuition: The Telephone Game

How it works:

- Each word updates the hidden state
- Hidden state = "memory" of what came before
- Pass memory to next step

The Problem:

- Like a game of telephone!
- Message gets corrupted over time

Message Length	Quality
10 words	Clear
50 words	Fuzzy
100 words	Lost!

> RNNs forget old information — the "vanishing gradient problem".

Further reading: LSTM/GRU cells help but don't fully solve this.

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?

Text: "Alice picked up the golden key. She walked to the door and tried to open it with the ___"

Approach	What It Sees	Limitation
Fixed Window	"with the"	Can only see last few words
RNN	Blurry summary	Memory degrades over time
ATTENTION	Any word directly!	"Let me check... 'key' was mentioned!"

> Attention is like having a **searchable index** over the entire text!

Attention: The Library Analogy

You're at the library looking for information.

Your Question (Query):
"What opens doors?"

You scan the shelves (Keys):

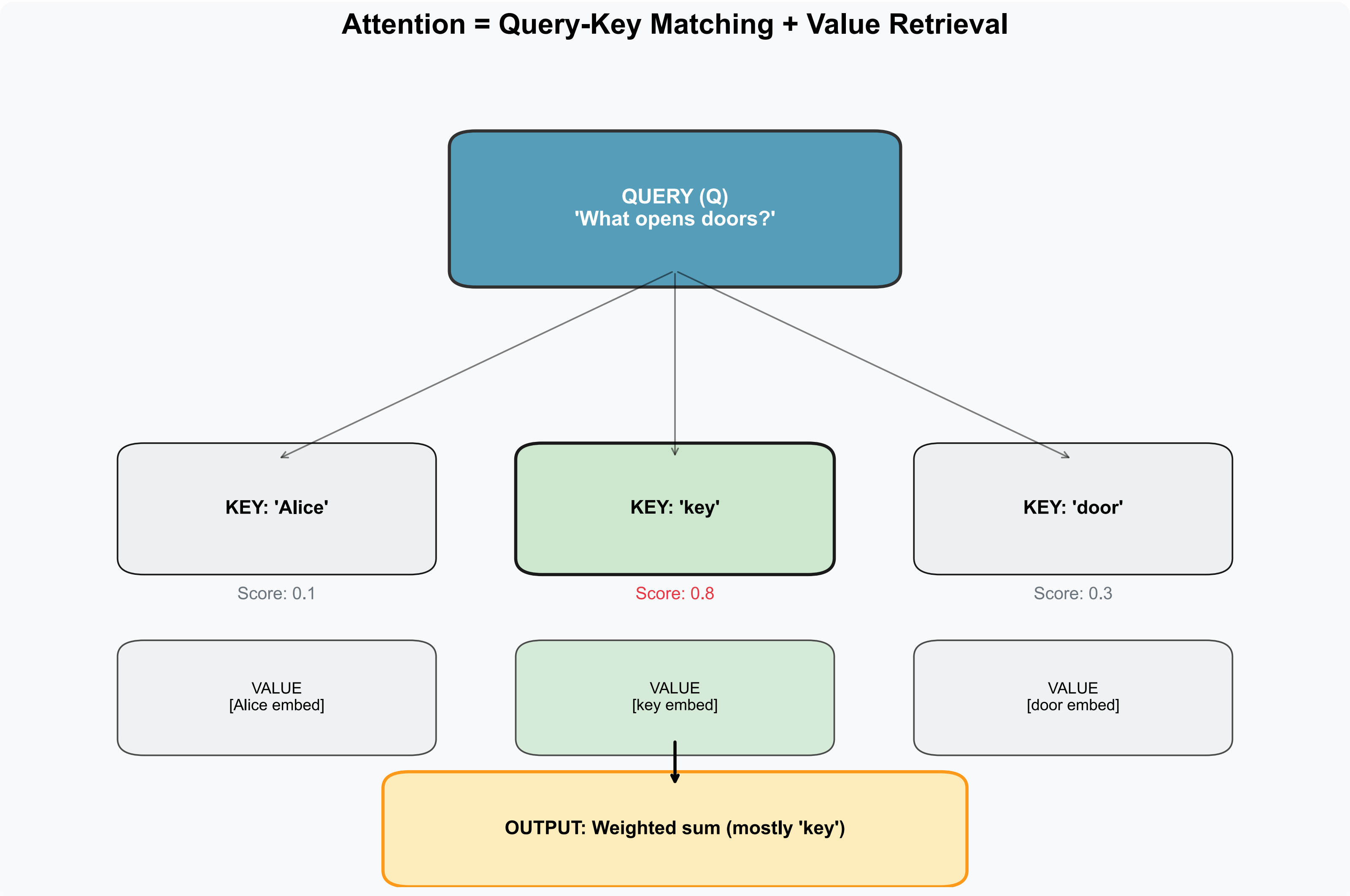
- "key" → Looks relevant!
- "door" → Somewhat related
- "Alice" → Not relevant

Book (Key)	Match Score
"key"	**0.8**
"door"	0.3
"Alice"	0.1

You read (Values) mostly from "key"!

Attention = Query → match with Keys → weighted sum of Values

Attention in Action



Why Attention is Powerful

Example: "The animal didn't cross the street because **it** was too ___"

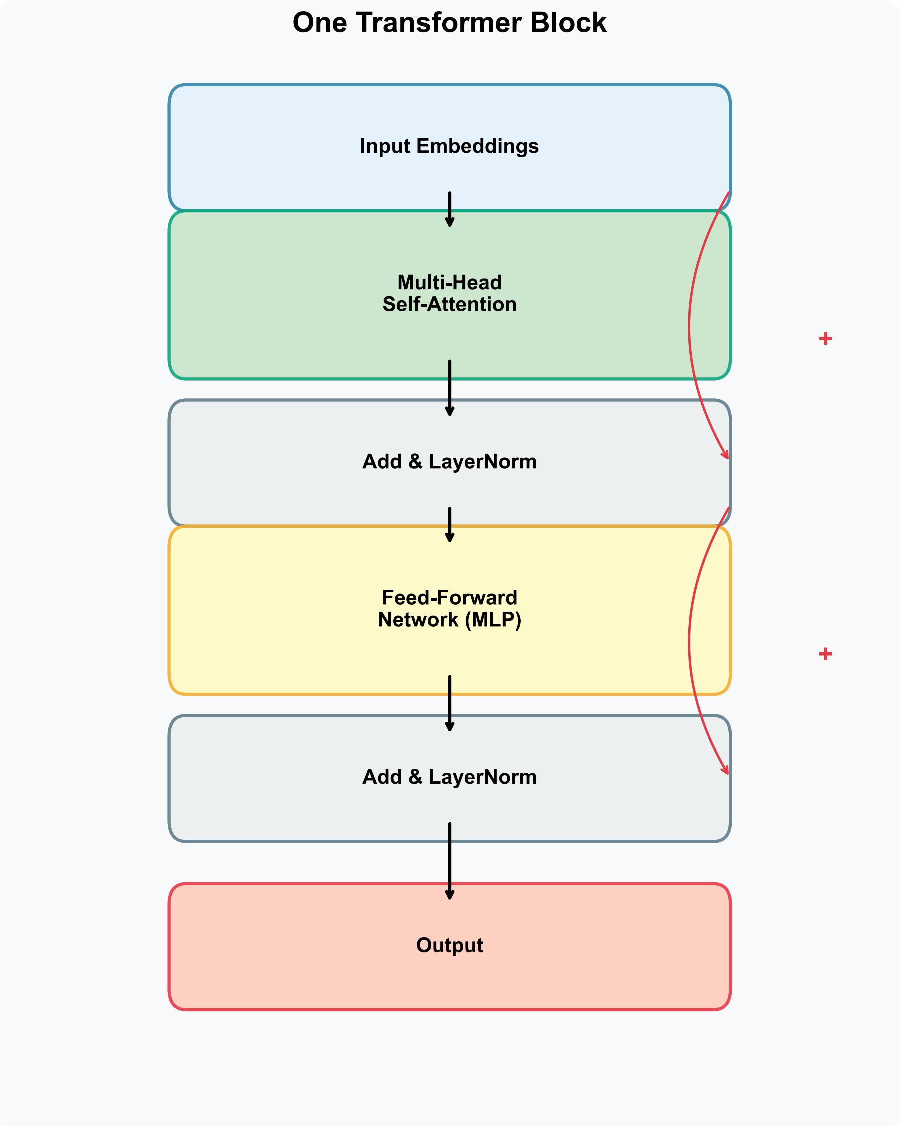
What does "it" refer to?

Word	animal	street
Attention Score	**0.75**	0.15

> The model learns to connect "it" to "animal" — this is called **coreference resolution**, learned automatically!

Further reading: Multi-head attention, self-attention matrices, positional encoding details

The Transformer: Putting It Together



The Two Key Components:

1. **Self-Attention**
   - "Who should I pay attention to?"
   - Every word looks at every word
2. **Feed-Forward Network**
   - Process the gathered information
   - Make predictions

Stack 96+ of these blocks!

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**.

Why Not Characters or Words?

LLMs don't process characters OR words — they use **TOKENS** (subwords).

Approach	Example	Problem
Characters	"hello" → ['h','e','l','l','o']	5 steps for one word! Too slow
Words	"hello" → ["hello"]	What about "unhappiness"? Millions of words!
Tokens	"unhappiness" → ["un", "happiness"]	Best of both! ~50K vocabulary

Tokenization Examples

Text	Tokens	Count
"Hello world"	["Hello", " world"]	2
"ChatGPT"	["Chat", "G", "PT"]	3
"unhappiness"	["un", "happiness"]	2
"don't"	["don", "'t"]	2
"2024"	["2024"]	1
"12345678"	["123", "456", "78"]	3

Common words = 1 token. Rare/long words = multiple tokens. Spaces often included: " world" not "world"

How BPE Works (Simplified)

Byte Pair Encoding — Start with characters, merge common pairs:

Step	Vocabulary	Example: "low lower lowest"
Start	All characters	l, o, w, e, r, s, t, ...
Merge 1	+ "lo"	"lo" appears often together
Merge 2	+ "low"	"low" appears often
Merge 3	+ "er"	"er" is common suffix
Merge 4	+ "est"	"est" is common suffix
Final	~50,000 tokens	Mix of chars, subwords, words

Common patterns become single tokens. Rare words split into pieces.

Why Tokenization Matters

LLM failures often trace back to tokenization!

Task	Problem	Why
"How many r's in strawberry?"	Often wrong!	"strawberry" → ["str", "aw", "berry"]
Math with big numbers	Inconsistent	"1234" = 1 token, "12345" = 2 tokens
Non-English text	Expensive!	"Hello" = 1 token, "नमस्ते" = 6 tokens
Counting characters	Hard	Model sees tokens, not characters

> The model doesn't "see" individual characters — it sees tokens!

Positional Encoding

How does the model know word ORDER?

Problem: Attention is permutation-invariant — "Dog bites man" and "Man bites dog" look the same!

Solution: Add position information to each embedding.

Component	Value	Example
token_embedding("cat")	[0.5, 0.3, 0.8, ...]	Word meaning
position_encoding(pos=3)	[0.1, -0.2, 0.4, ...]	Position info
final_embedding	[0.6, 0.1, 1.2, ...]	Sum of both

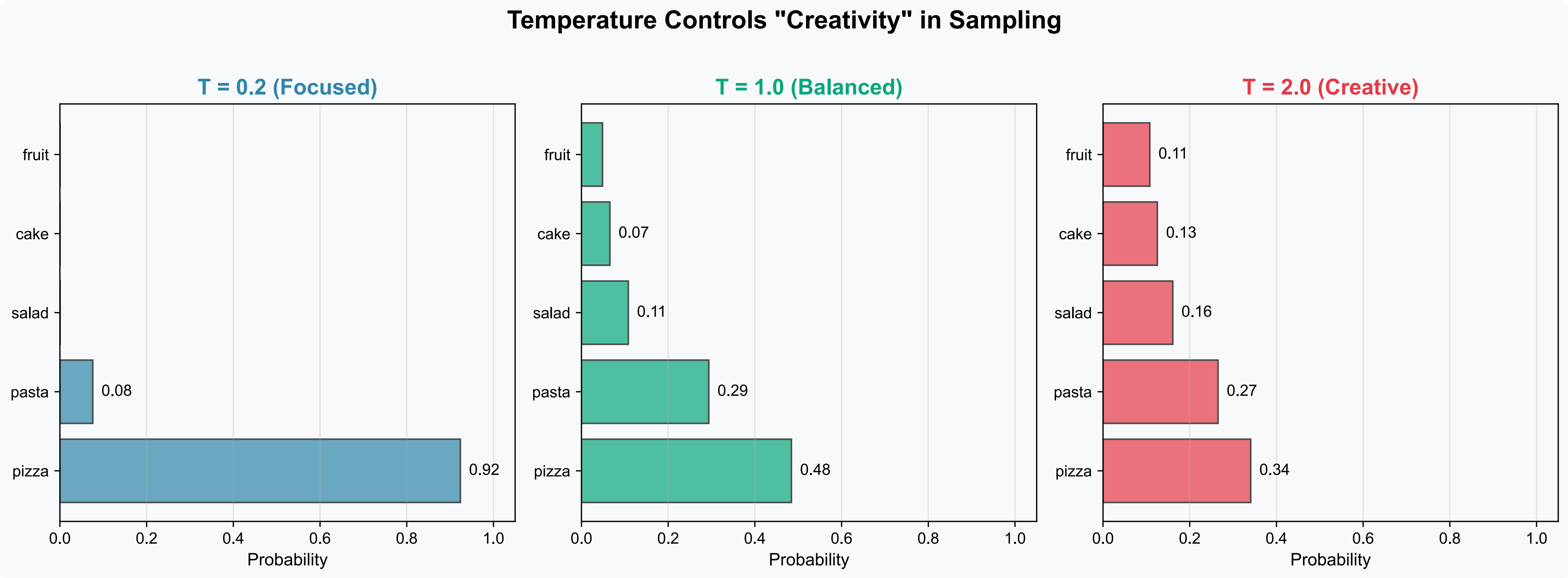
Now "cat" at position 3 ≠ "cat" at position 10!

Original formula: $PE_{(pos,2i)} = \sin(pos/10000^{2i/d}), PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$

Modern models: **Learn** position embeddings!

Temperature: The Creativity Knob

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



- **Low temp** → Always picks the most likely (boring but safe)
- **High temp** → Spreads probability more evenly (creative but risky)

Temperature: The Math

$$\text{probs} = \text{softmax}(\text{logits}/T)$$

Example logits: [2.0, 1.0, 0.5, 0.1]

Temperature	Probabilities	Effect
T = 1.0 (normal)	[0.43, 0.26, 0.19, 0.12]	Balanced
T = 0.1 (cold)	[0.99, 0.01, 0.00, 0.00]	Almost deterministic
T = 2.0 (hot)	[0.32, 0.27, 0.22, 0.19]	More uniform/random
T = 0 (greedy)	argmax	Always pick highest

Lower temperature → more focused; Higher temperature → more creative/random

Other Sampling Strategies

Besides temperature, there are other ways to control generation:

Strategy	How it Works	When to Use
Top-K	Only consider top K tokens	Simple, predictable
Top-P	Include tokens until cumulative prob > P	Adaptive, popular
Greedy	Always pick highest probability	Deterministic, boring

Example with Top-3:

- All options: pizza(0.4), pasta(0.3), salad(0.1), shoes(0.05)...
- Top-3 only: pizza(0.53), pasta(0.40), salad(0.07) ← renormalized

> Top-P (nucleus sampling) is adaptive: fewer options when confident, more when uncertain!

Why Sampling = Different Outputs

Same prompt, different runs → different text!

Run	Prompt	Generated
1	"The cat"	"The cat sat on the mat."
2	"The cat"	"The cat jumped over the fence."
3	"The cat"	"The cat was sleeping peacefully."

Each token is **sampled** from a probability distribution.

Different random samples → different paths → different outputs!

Training at Scale: GPT-3 Numbers

Aspect	GPT-3
Parameters	175 billion
Training data	500 billion tokens
Data sources	Common Crawl, Books, Wikipedia
Training time	~1 month
Hardware	10,000 GPUs
Cost	~$4.6 million (electricity alone!)

> The core algorithm is simple. The scale is what makes it work!

From GPT to ChatGPT: 3 Training Stages

Stage	Goal	Data	Result
1. Pre-training	Learn language	Internet text (trillions of tokens)	Base model (can complete text)
2. SFT	Follow instructions	(instruction, response) pairs (~100K)	Follows directions
3. RLHF	Be helpful & safe	Human preference rankings	ChatGPT!

Stage 1: Pre-Training

Goal: Learn patterns from massive text

Input	Target	What model learns
"The capital of France"	"is"	Geography facts
"def hello():"	"print"	Python syntax
"To be or not to"	"be"	Shakespeare

- Cost: **$10M-$100M+**
- Time: **Months**
- Result: Can complete text, but not helpful or safe

Stage 2: Supervised Fine-Tuning (SFT)

Goal: Learn to follow instructions

User Says	Model Should Say
"Explain photosynthesis simply"	"Plants use sunlight to make food from air and water!"
"Write a haiku about coding"	"Fingers on keyboard / Logic flows through the machine / Bugs hide in plain sight"
"Summarize this article"	[Actual summary]

- Data: ~100K human-written examples
- Cost: **~$1M**
- Result: Follows instructions, but can still be harmful

Stage 3: RLHF (Alignment)

Goal: Learn human values — be helpful, harmless, honest

Step	What Happens
1. Generate	Model produces 3 responses to same prompt
2. Rank	Humans say: Response A > B > C
3. Reward Model	Train a model to predict rankings
4. Optimize	Use RL to maximize reward

Why needed? SFT models can still:

- Follow dangerous instructions
- Make up facts confidently
- Be technically correct but useless

Result: ChatGPT = Pre-training + SFT + RLHF

Summary: The Full Stack

Layer	Component	Purpose
0	The Task	Predict $P(\text{next} \mid \text{context})$
1	Representation	Tokens → Embeddings (meaning as vectors)
2	Context	Self-Attention (look at relevant past tokens)
3	Computation	Feed-Forward layers (process information)
4	Stacking	Repeat attention+FFN 96 times for depth
5	Training	Next token prediction on internet-scale data
6	Alignment	Instruction tuning + RLHF for helpfulness

Key Takeaways

The 5 Big Ideas

#	Idea	Key Insight
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 Big Reveal

All of ChatGPT, Claude, Gemini...

...is just predicting the next token, really well.

> One simple idea, scaled massively: Given context, what comes next?

Questions?