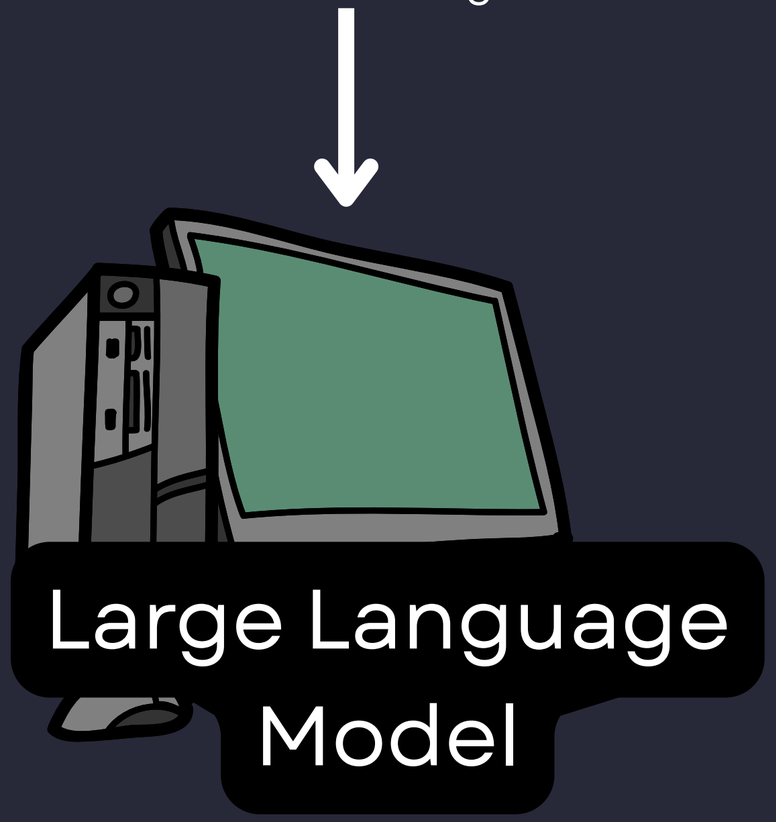
LLMs and transformer models

What they do and how to use them better

Brendan Beh
AI.SEA

LLMs are probability machines

User:
What is the meaning of life?



The	Meaning	Guide
Well	Hitchhiker's	Walking
Why	Point	Route
Nobody	Way	Shoes
To	Reason	Path
Life	Cause	Trail

LLM:
"The Hitchhiker's Guide To The Universe tells us that the answer is 42."

Each sentence gives a probability map about what comes next

We are out of eggs. I am heading to the ___

Context matters a lot - it defines probabilities for the output

We are out of eggs. I am heading to the ___

If we don't capture enough context, we can't form good sentences or extract info properly.

Older AI models struggled to capture long-range context

We are out of eggs. I am heading to the ___

We are out of eggs. I am heading to the supermarket because ___

We are out of eggs. I am heading to the supermarket because they sell ___

We are out of eggs. I am heading to the supermarket because they sell very nice ___

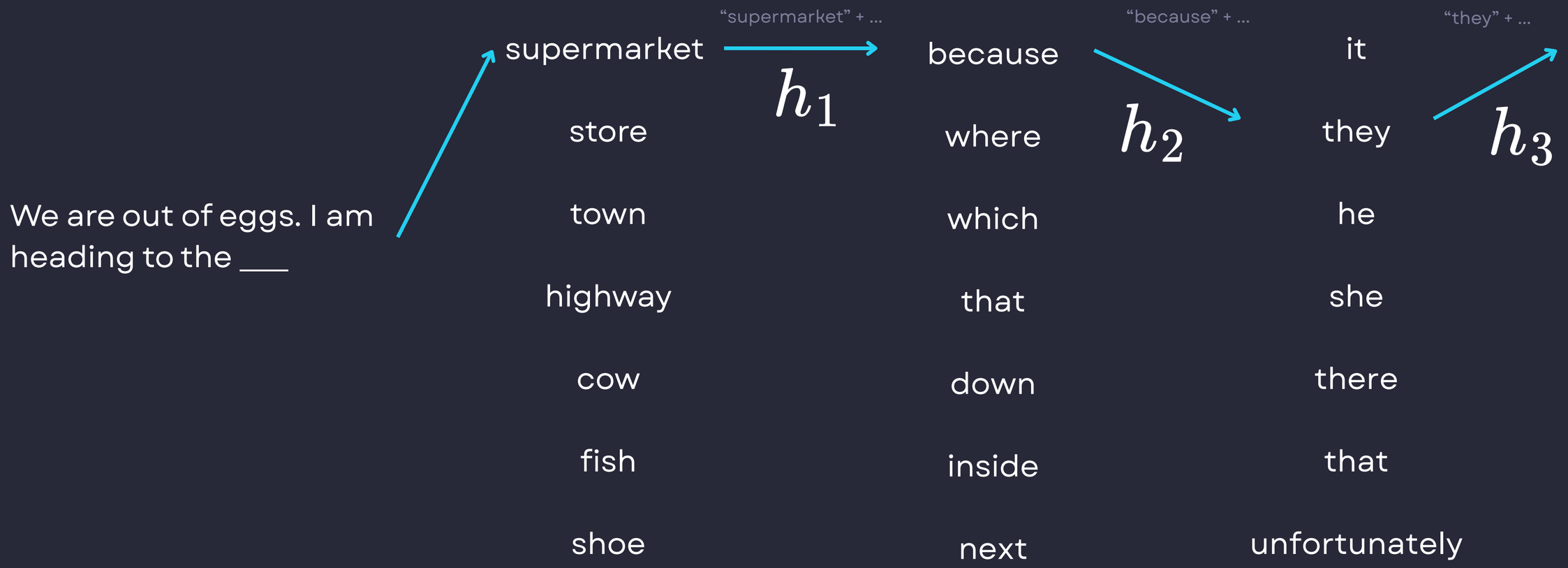
We are out of eggs. I am heading to the supermarket because they sell very nice fresh fish.

Older models e.g. RNNs took inputs sequentially and used hidden states to capture past context.

However, older info gets diluted over time.

This meant long dependencies were difficult to capture.

States were updated sequentially - this overloaded long-range dependencies



LLMs let each word “talk” to each other to figure out which is relevant to the context

Importantly, this is done in parallel - this enables the model to take information from much further away.

We need to first convert words to numbers so we can process the sentence

Everybody who has eaten at Taco Bell has ___

An input is first broken up into **tokens.**

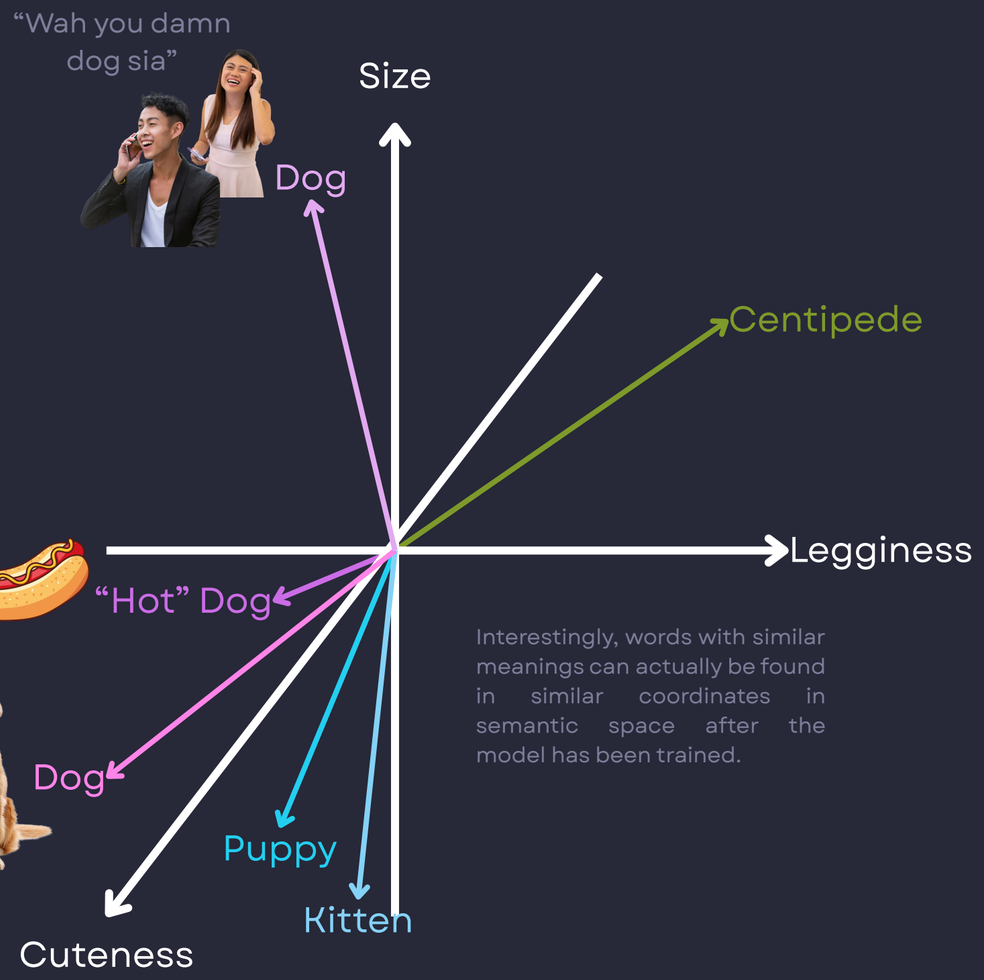
Image from Georgia Tech Polo Club

These tokens are encoded into a "**vector**" (i.e. a list of numbers) by the neural network.

This vector encodes the "meaning" of the token + its position in the input.

This is called the **embedding** process.

Vectors can be thought of as coordinates in “meaning” space



Interestingly, words with similar meanings can actually be found in similar coordinates in semantic space after the model has been trained.

The process of converting a word into a vector is done through an **embedding** matrix.

$$W_E = \begin{pmatrix} 1.0 & -2.4 & 9.8 & 7.5 & .. & -6.2 & 3.6 \\ -2.5 & -4.4 & -8.7 & 9.6 & \dots & -3.1 & 8.8 \\ -7.3 & 3.5 & 9.3 & -4.4 & \dots & 2.5 & 8.3 \\ \dots & & & & & & \\ 3.2 & 2.4 & -0.6 & 2.5 & \dots & 7.3 & -0.7 \end{pmatrix}$$

The height of this matrix = the no. dimensions in the “meaning” space

The length of this matrix = the number of token permutations
(Think of as the no. words in the dictionary)

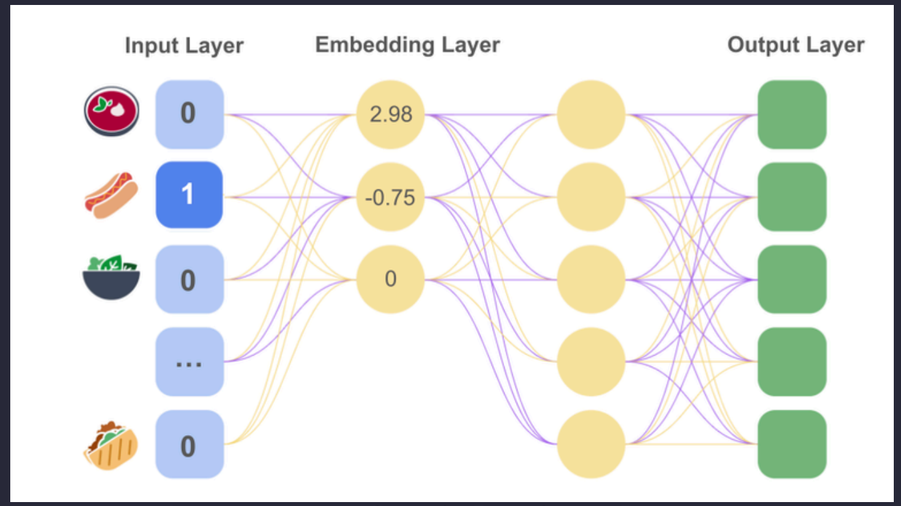


Image source: Google Developer Program

Make it easy for your embeddings to work

Avoid emoji/rare unicode in critical spans

Normalize input where possible.

Watch token counts; prefer concise wording and short identifiers.

“customer-complaint-resolution-protocol-v7” → “CCR-v7”.

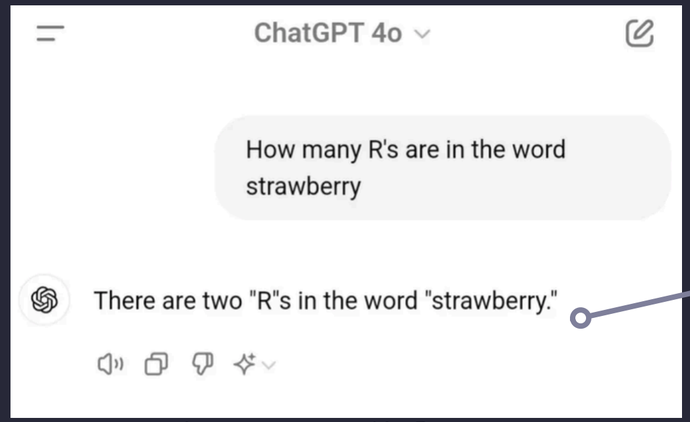
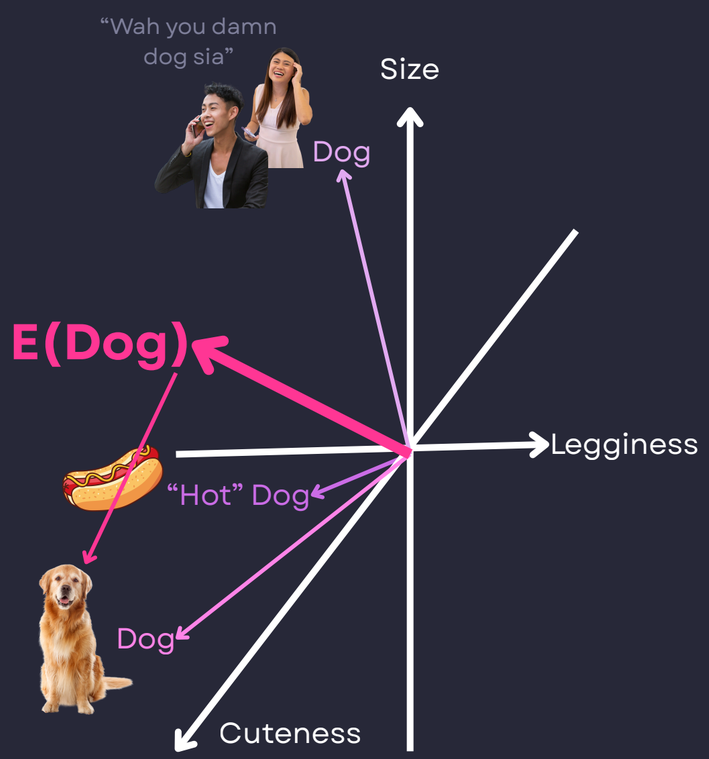


Image source: 16x Prompt

Image source: 16x Prompt

We need to use context to identify the correct meaning



E(Dog) - “Size” + “Cuteness” + “Legginess” ~ Dog (Animal)

How do we find the correct “meanings” (vectors) to add to get to the right answer?

We need each word to “ask” other words in the input what their relationship is.

This is done by a neural network called a **transformer**, using a mechanism called the **attention**. Essentially, we will be asking

"How much does each other token of input affect the interpretation of this token?"

The core of Attention is the Query-Key-Value mechanism

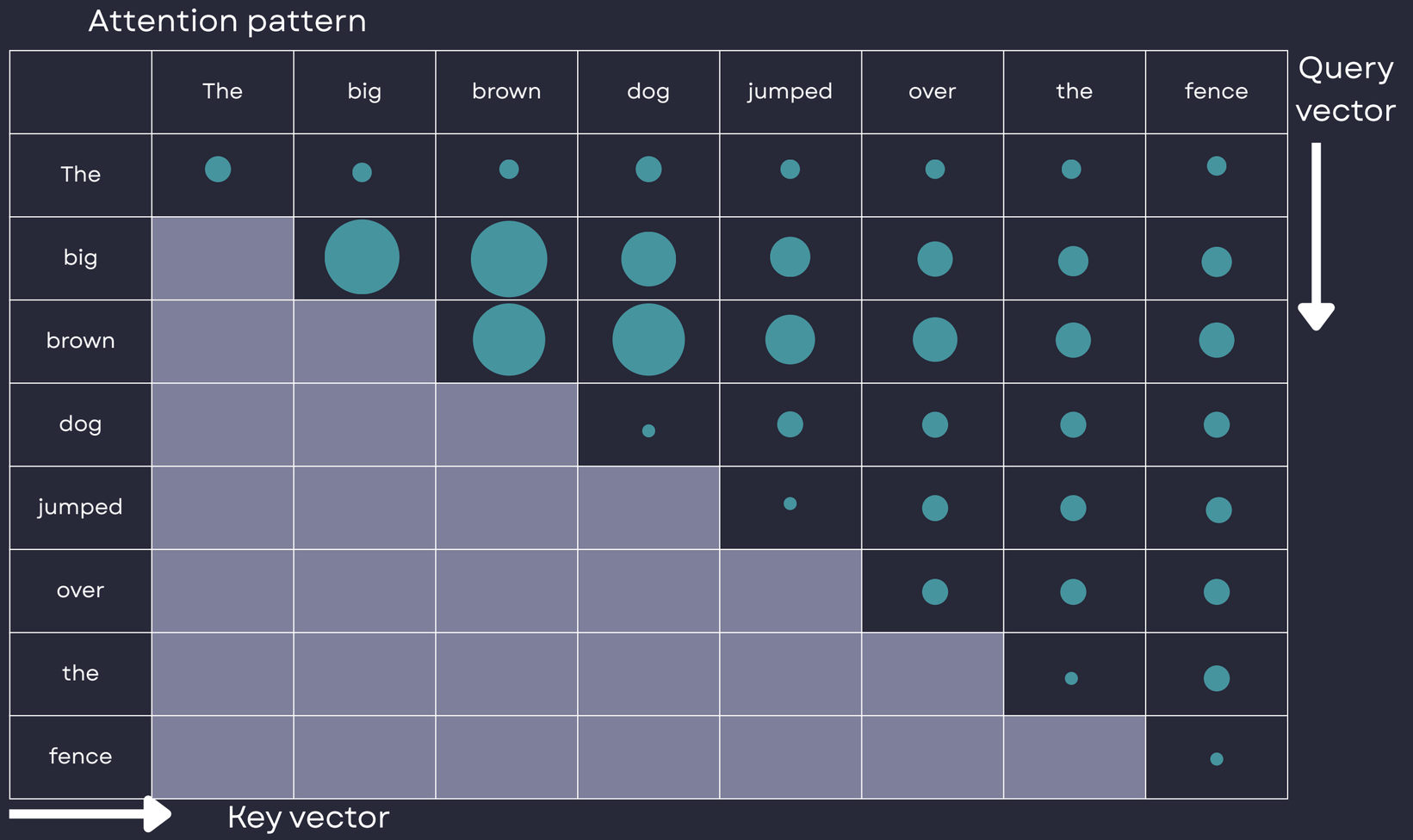
Query (Q): “Is there an adjective before this word?”

Key (K): “Which of these words is an adjective and where is it?”

Value (V): “What is the adjective we are looking at?”

Each query is a single head of attention. LLMs run multiple queries to output an answer (multi-head attention).

“The big brown dog jumped over the fence.”



Attention in formal terms

$$\text{Attention}(\text{Q},\text{K},\text{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

$$= \text{softmax}\left(\frac{\mathbf{x}\mathbf{Q}\mathbf{K}^T\mathbf{x}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

This acts as a (normalized) dot product to find the overlap between the transformed Query-Key pair for two given tokens.

The probabilities are rescaled by the Value, i.e. the “meaning”. The model views the Value vector for a word as providing the piece of information that "should be added to the embedding of that something else in order to reflect this" relevant context.

$$\mathbf{K} = \mathbf{x}\mathbf{W}_k$$
$$\mathbf{V} = \mathbf{x}\mathbf{W}_v$$
$$\mathbf{Q} = \mathbf{x}\mathbf{W}_q$$

To ask a question / generate an answer is to act upon the input vector by a smaller-dimensional matrix.

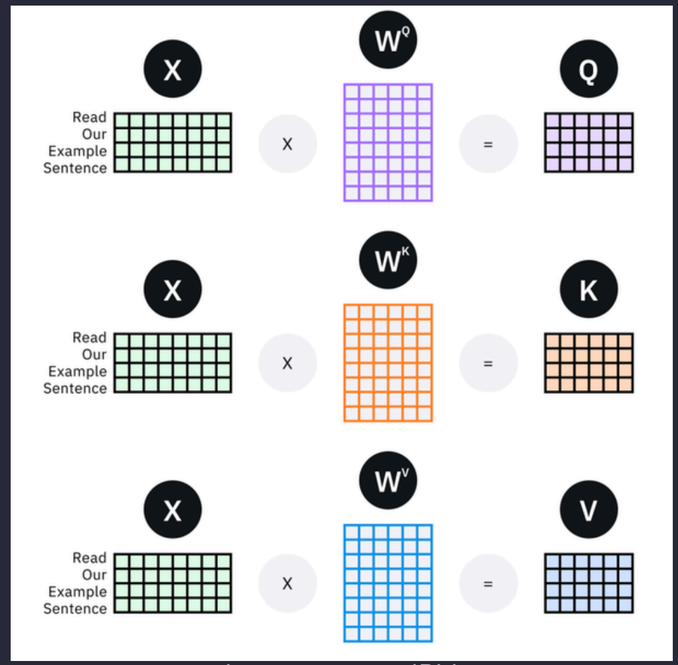


Image source: IBM

QKV matching loves structure - help the LLM by giving it a repeatable format

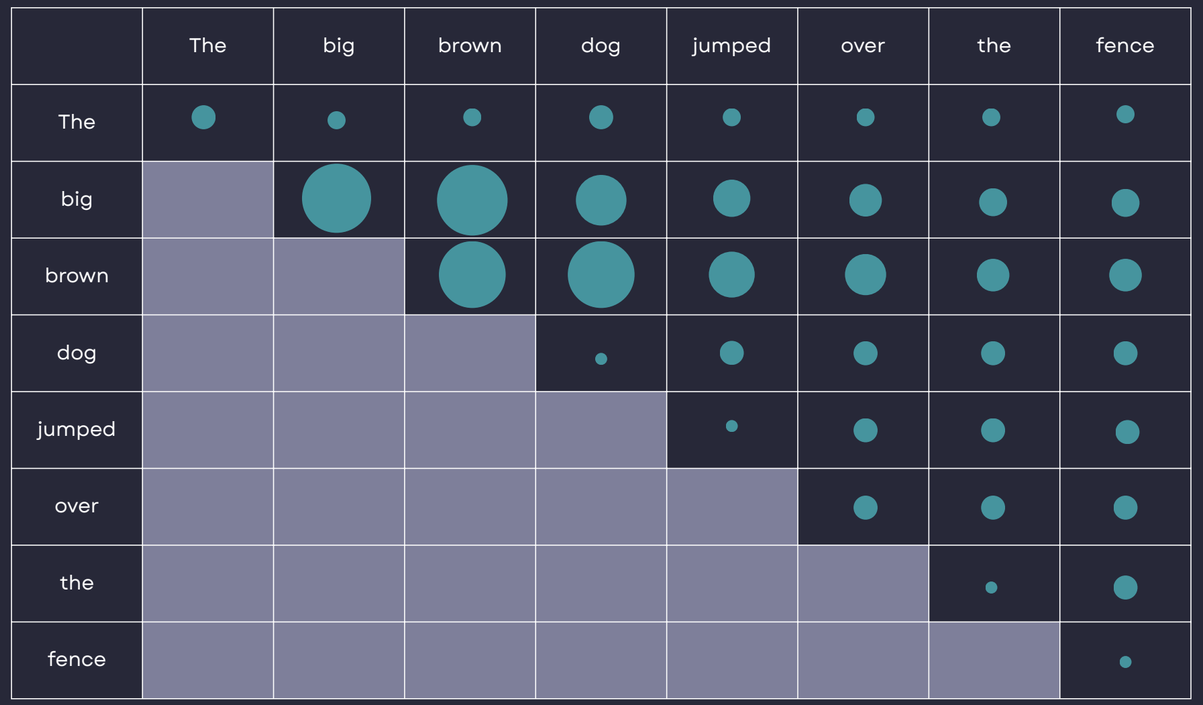
Use strong delimiters (###, triple backticks, YAML blocks) to isolate sections.

- Wrap context with ### CONTEXT / ### END CONTEXT.
- Put few-shot immediately before the query; mirror output schema exactly.
- Use explicit labels (“Instruction:”, “Examples:”, “Your turn:”) to produce distinctive keys.

```
System: You are a tax summarizer. Output JSON schema below.
### SCHEMA
{ "income": number, "deductions": number, "summary": string }
### EXAMPLES
Input: ...
Output: {"income":..., "deductions":..., "summary":"..."}
### YOUR TURN
Input: ...
Output:
```

Take note of your input prompts!

Attention pattern



The attention scales as $\mathcal{O}(N^2)$
Where N is the size of the input vector.

The effectiveness of attention degrades as the total input length increases

Rather than long system prompts:

- Pre-segment long docs (titles + abstracts + chunks)
- Replace long boilerplate with variables (“{policy}”, “{schema}”).
- Use headlines, bullet points, and IDs instead of verbose prose.
- For file intake, chunk with small overlaps (e.g., 200–400 tokens, 10–15% overlap) and index via embeddings.
- Prefer tool calls (deterministic APIs) over verbose “reasoning” when possible.

To ensure we don't look too far ahead, we mask later tokens to keep information flowing one way

"The big brown dog jumped over the fence."

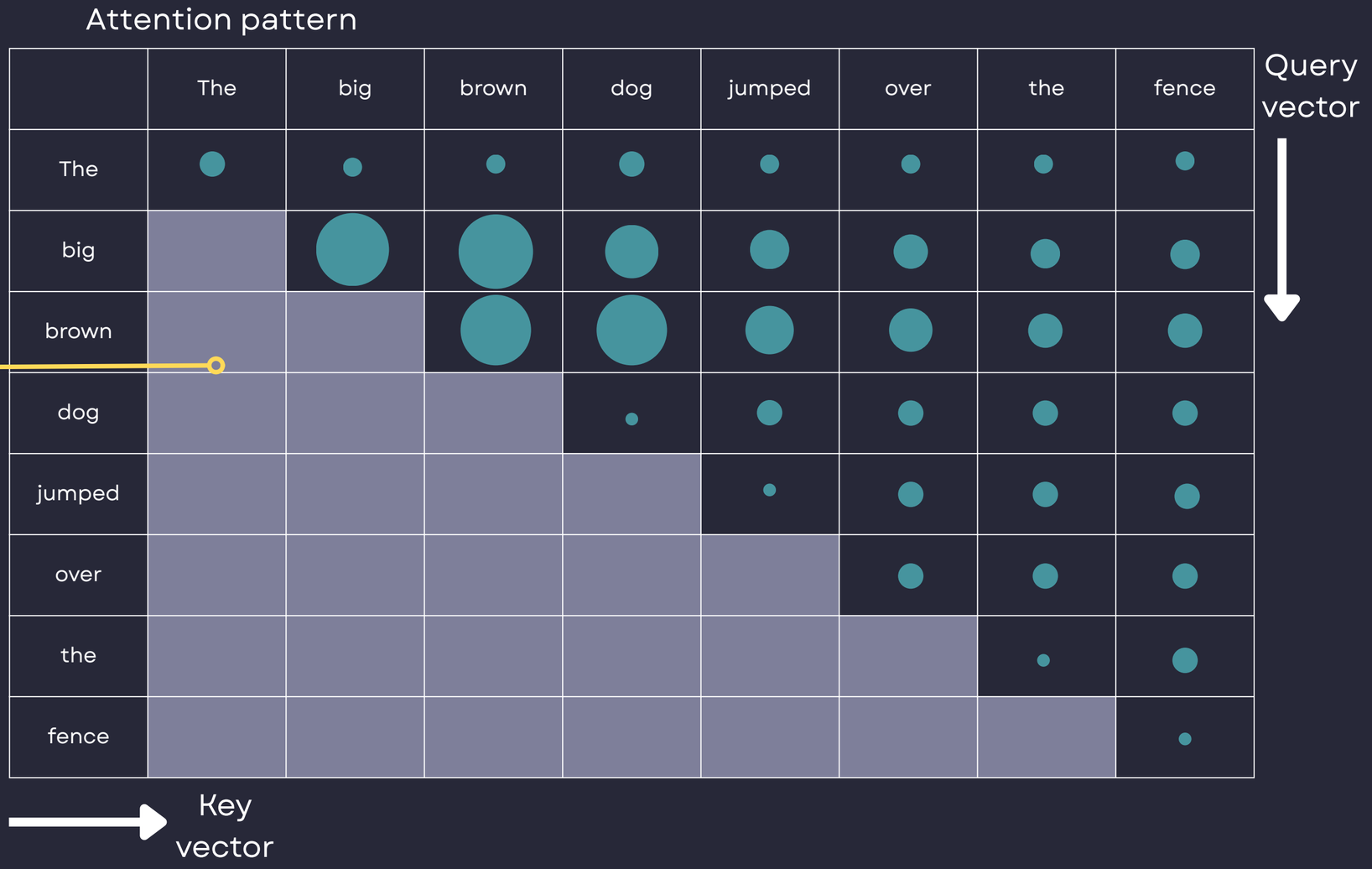
E.g. The big brown dog...

If we predict too far down the chain (...stole a bagel...) this can influence the training and results.

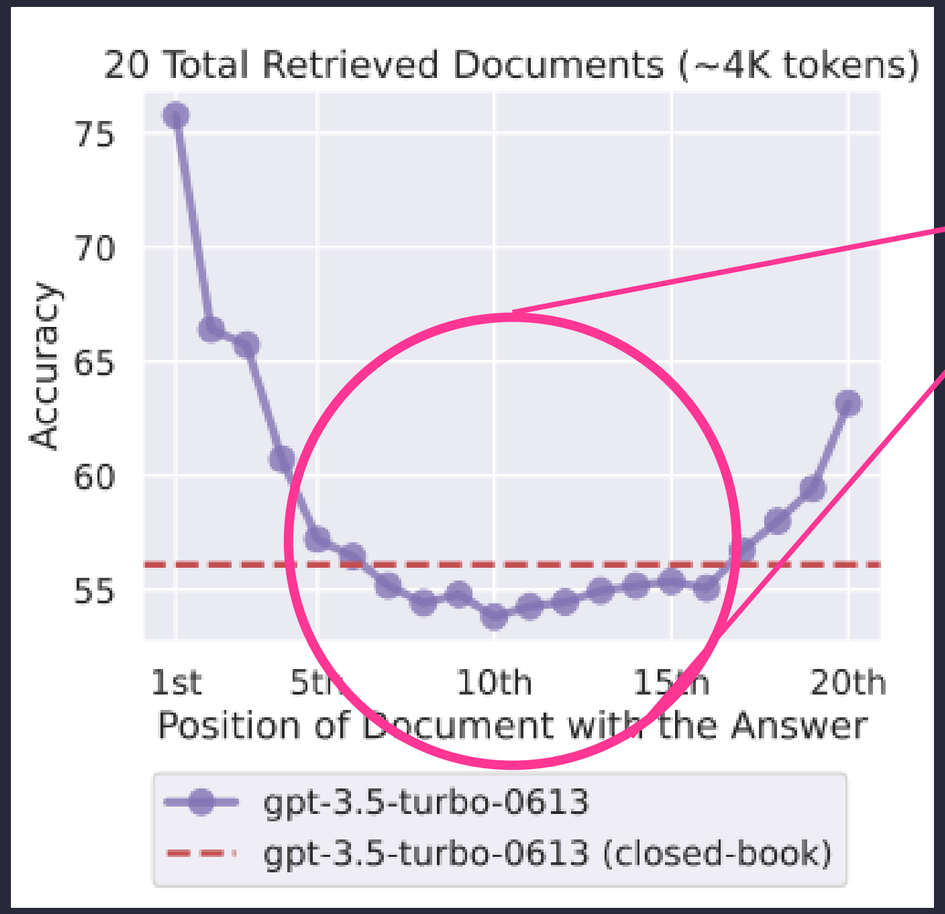
Therefore we don't want later tokens to influence earlier ones.

We can manipulate the attention function to make sure only earlier tokens can affect later ones by adjusting the Q-K overlap to zero.

This is known as **masking**.



The lost-in-the-middle phenomenon



While attention should (in theory) be position independent, in practice they tend to prioritise the head and tail of the system prompt.

This is the "lost-in-the-middle" phenomenon.

Why does this happen?

- Causal masking tends to inherently bias attention toward earlier sequence positions in deep networks
- Tokens in deeper layers have increasingly contextualized views of earlier tokens, amplifying the influence of initial positions regardless of semantic content
- LLMs are trained on data that reflects the human primacy-recency effect (which naturally favors recall of information presented at the beginning and the end of a sequence), developing analogous biases

How you arrange your prompt matters!

Prompt spine:

- Role + hard rules (top)
- Minimal context
- Task statement
- Examples (few-shot)
- Restate critical constraint (bottom)

Use compact bullets, numbered steps, and one sentence per rule.

Example:

- Top: “You are a strict validator. Output JSON only.”
- Context (3–5 bullets max)
- Task
- Bottom: “Reminder: JSON only, no prose.”

How you arrange your prompt matters!

Prompt spine:

- Role + hard rules (top)
- [illegible]
- [illegible]
- [illegible]
- Restate critical constraint (bottom)

Use compact bullets, numbered steps,
and one sentence per rule.

Example:

- Top: “You are a strict validator.
- Bottom: “Reminder: JSON only, no prose.”

- Front-load critical instructions; end-load key reminders.
- Keep prompts concise; avoid burying requirements in the middle.
- Repeat the one most critical constraint at the end.

Summary

Condense and sanitize your system prompts:

- Pre-segment long docs (titles + abstracts + chunks)
- Replace long boilerplate with variables (“{policy}”, “{schema}”).
- Use headlines, bullet points, and IDs instead of verbose prose.
- For file intake, chunk with small overlaps (e.g., 200–400 tokens, 10–15% overlap) and index via embeddings.
- Prefer tool calls (deterministic APIs) over verbose “reasoning” when possible.
- Remove decorative emojis and abbreviate where possible.

System prompt arrangement matters, avoid the LITM problem:

- Front-load critical instructions; end-load key reminders.
- Keep prompts concise; avoid burying requirements in the middle.
- Repeat the one most critical constraint at the end.
- Use strong delimiters (###, triple backticks, YAML blocks) to isolate sections.