COMS4995W32
Applied Machine Learning

Dr. Spencer W. Luo

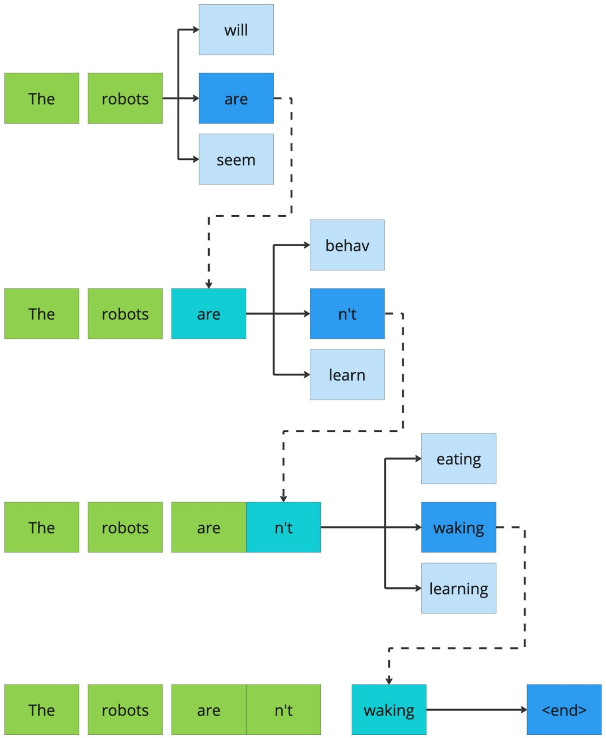
Columbia University | Fall 2025

Transformer Fundamental

Recap on MLP Colab

Motivation

How does ChatGPT work in a nutshell? 🤔



Task: Next-Token Prediction 🎯

Task: given previous tokens $x_1 \ldots x_{t-1}$, predict the next token x_t

The model outputs a probability distribution:

$$\max p(x_t \mid x_1 \ldots x_{t-1}) \in R^{(|V|)}$$

where $|V|$ = vocabulary size

Each prob represents how likely each token is to be the next one

Essentially a multi-class classification problem at each position

Tokens 🧱

Text is split into tokens

- Often subwords, not full words
- Better discrete units that capture reusable language patterns
- e.g., “playing” → “play” + “##ing”

Each token is mapped to an integer ID via the tokenizer’s vocabulary

- Usually between 10K - 50K tokens

Training Objective - Cross-Entropy Loss 📉

Since this is a multi-class problem, we use cross-entropy loss:

$$L = -\sum \log p(x_t \mid x_1 \ldots x_{t-1})$$

It penalizes the model when it assigns low probability to the correct next token

Same loss as we used in earlier MLP sessions

- Only difference: inputs are sequential texts

Predict the next token

- This simple rule is the core principle of Language Modeling

First Attempt: Use an MLP 🧩

Feed the entire context representation into an MLP

MLP learns non-linear mappings, and predicts next tokens

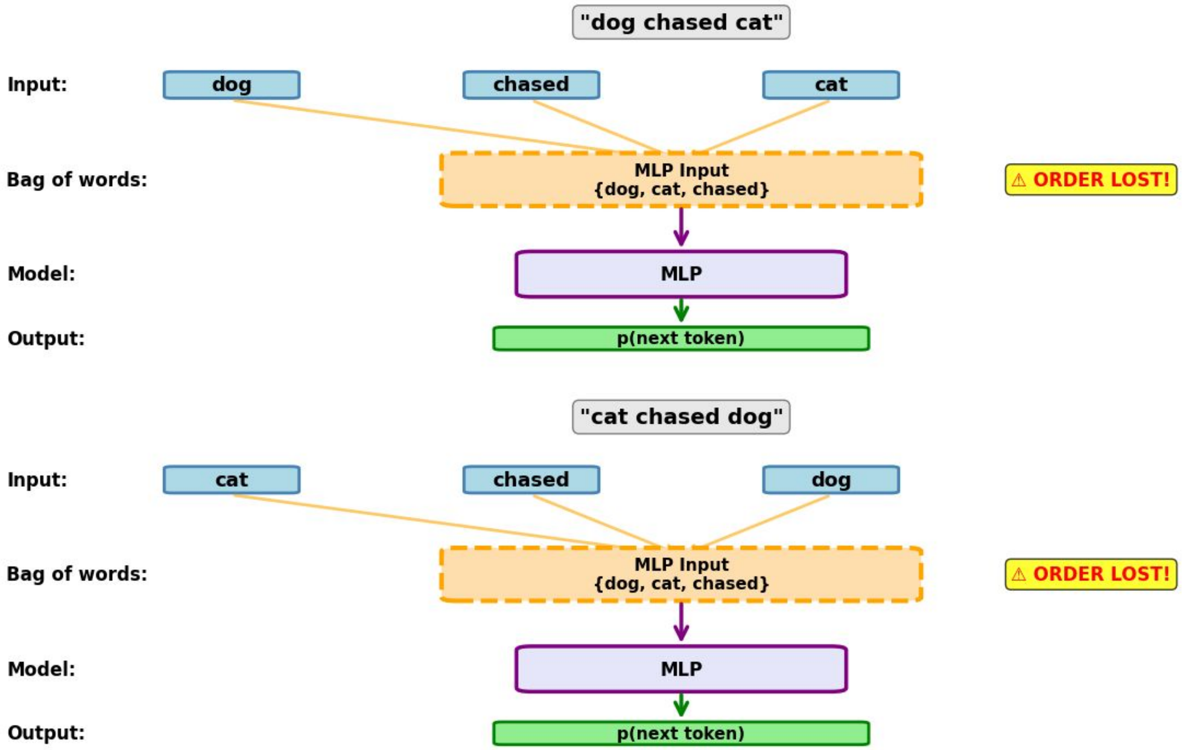
Pros ✅

- Learns flexible patterns and interactions between features

Cons ❌

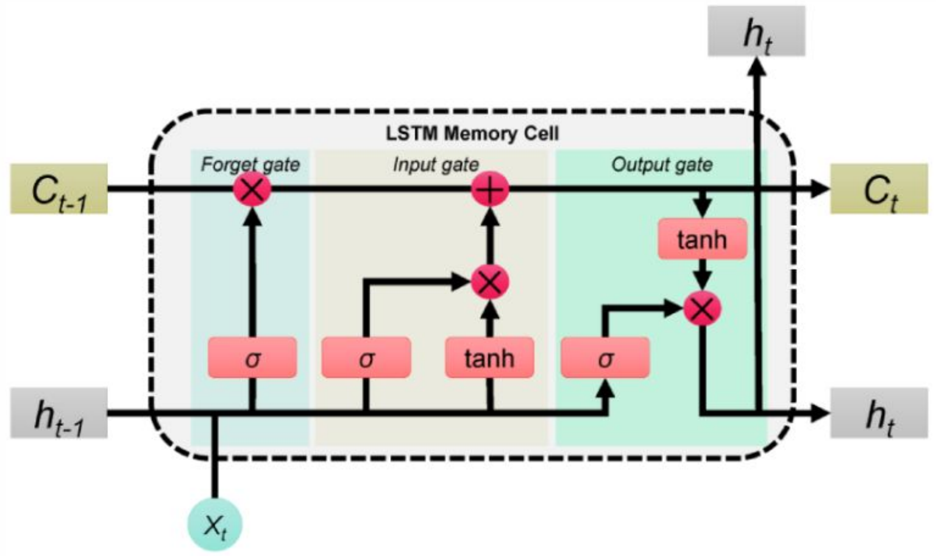
- Treats each token independently - *ignores order and position*
- No awareness of sequence or dependency between words

Order Matters



Add Order: Try RNN / LSTM 🚀

Recurrent Neural Networks / Long Short-Term Memory



Add Order: Try RNN / LSTM 🚀

Pros ✅

- Capture short-term context
- Great for small sequences

Cons ❌

- Training is slow and non-parallelizable
- Long-range dependencies fade (vanishing gradients)
- Inefficient for large-scale texts

What We Really Need ⚡

A neural network model that:

- Understands relationships between all tokens in a sequence
- Computes these relationships in parallel
- Learns which tokens matter most for the prediction

To Build Transformer 🤖

1️⃣ Self-Attention Mechanism

2️⃣ Residual Connection

3️⃣ Transformer Block

Self-Attention Mechanism

Motivation: Learning to Focus 🔎

Humans do not process all words equally - we focus on what matters

- When reading, your eyes naturally skip unimportant words and linger on key ones

In next-token prediction, not all previous words are equally useful

- Some words strongly influence what comes next, others can be safely ignored

We want the model to assign higher weights to the most relevant context words, when predicting next tokens

This idea - learning where to focus - is the foundation of Attention

Self-Attention ➡️

In next-token prediction, all the words come from the same sequence

Each word simply looks at the others in the same sentence to understand their contextual relationships

- This is called Self-Attention

It helps information flow among all words, allowing the model to understand context before predicting the next token

Core Flow 📄

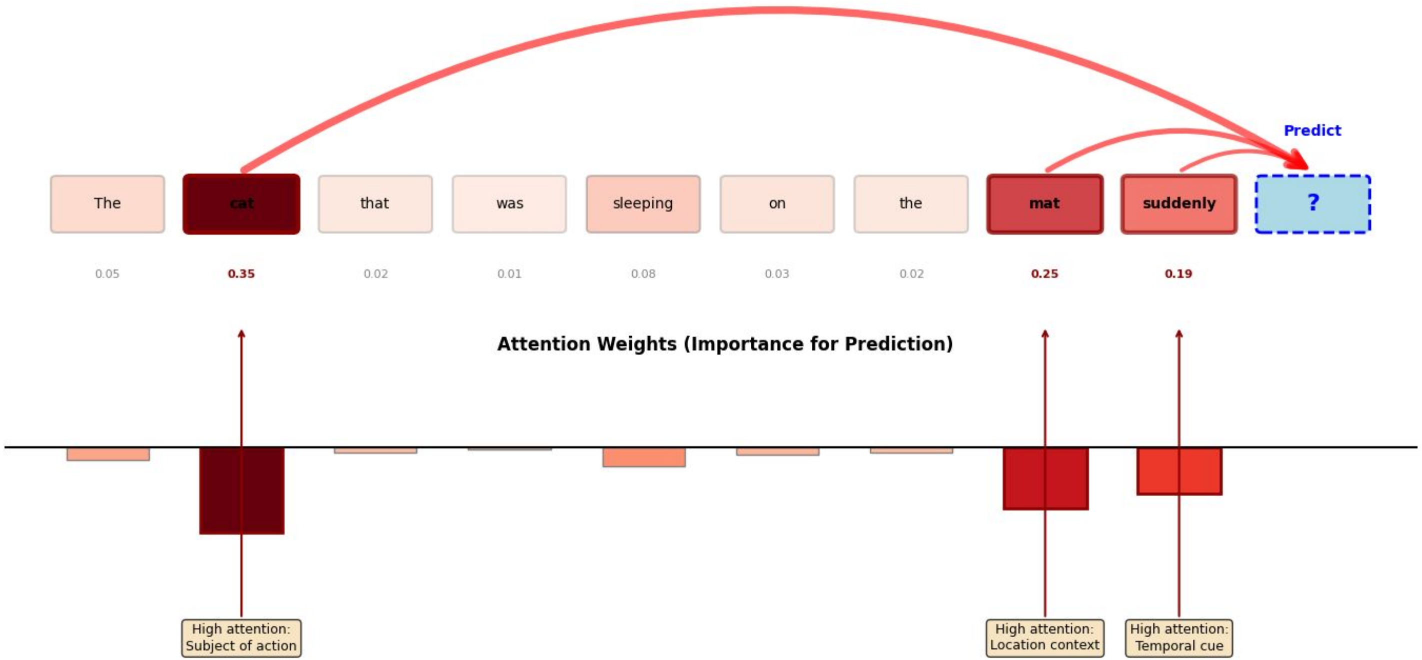
Each token asks: “Which other tokens should I pay attention to?”

The model computes similarity scores between tokens

These scores become weights that determine how much information to take from each token

The next token = weighted average of all other tokens’ information

Visualization 👀



Why Self-Attention Matters 🤔

Self-Attention allows information to flow among all words in a sentence

Each word updates its understanding based on what others mean

This creates rich, context-aware representations for every position

In next-token prediction, this helps the model understand the full context before guessing the next word

How does this actually happen inside the model?

Step 1 – Token Embeddings 🧱

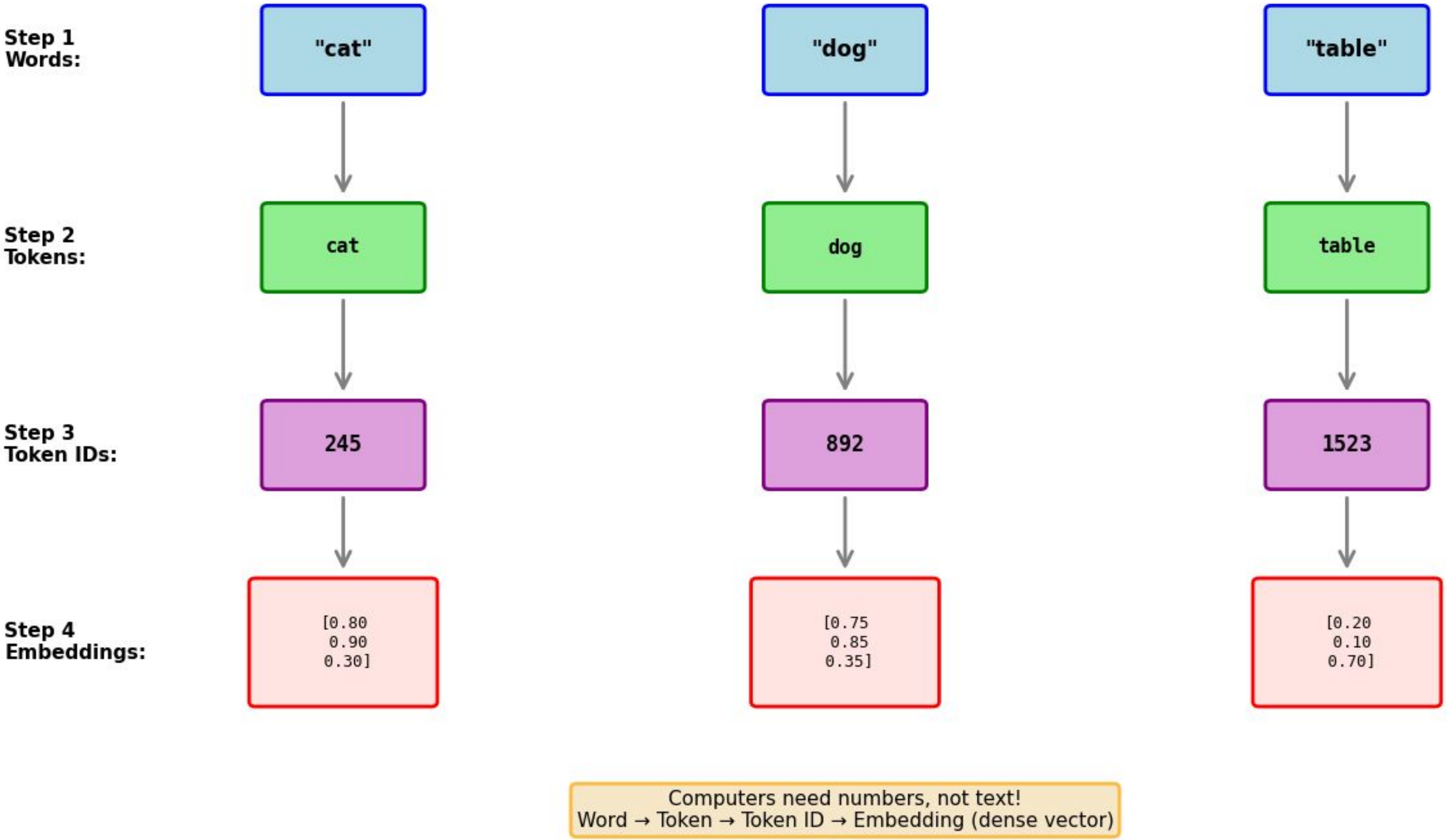
Computers cannot understand text directly

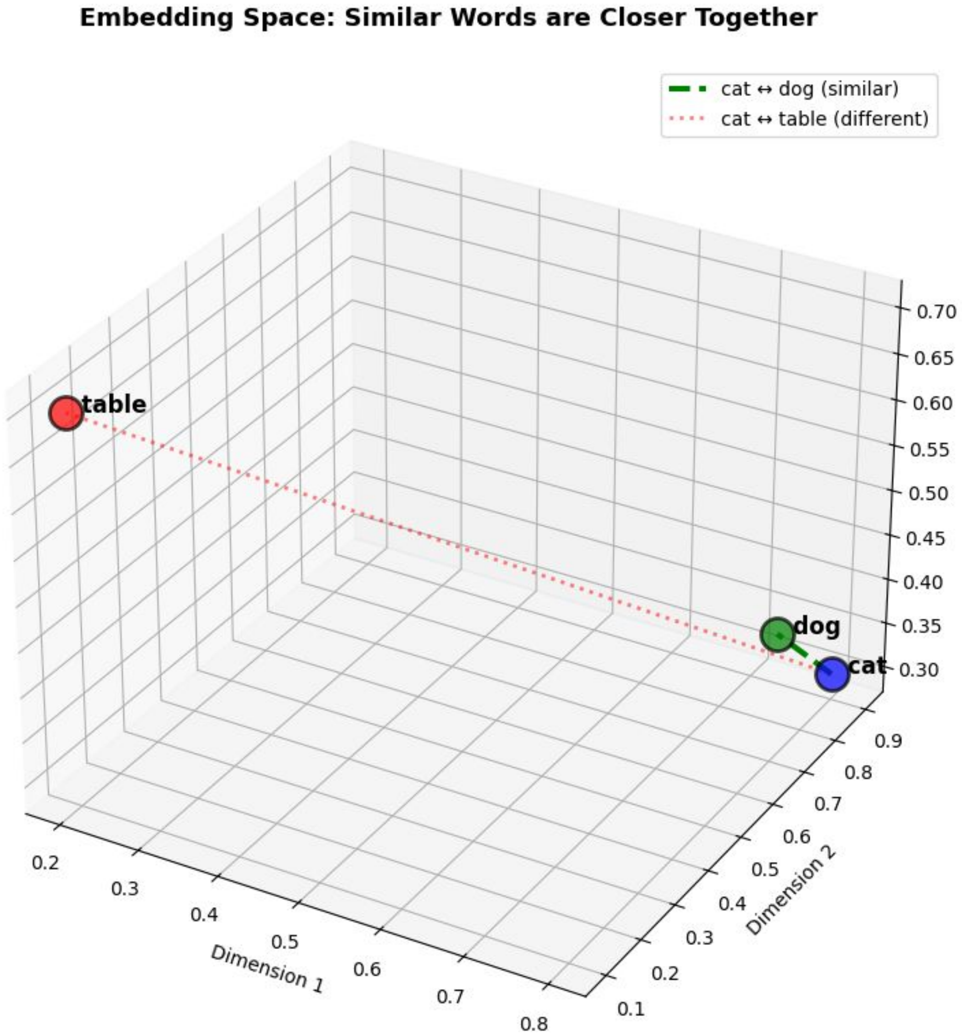
So each token (like “dog”, “cat”, “run”) is converted into a vector of numeric values

Ideally, these vectors shall capture semantic meaning

- Vectors of “cat” and “dog” are closer than the ones of “cat” and “table”

This vector form is called an embedding


Step 1
Words:
Step 2
Tokens:
Step 3
Token IDs:
Step 4
Embeddings:
"cat"
"dog"
"table"
cat
dog
table
245
892
1523
[0.80 0.90 0.30]
[0.75 0.85 0.35]
[0.20 0.10 0.70]
Computers need numbers, not text!
Word → Token → Token ID → Embedding (dense vector)


Embedding Space: Similar Words are Closer Together
cat ↔ dog (similar)
cat ↔ table (different)
table
dog
cat
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Dimension 1
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Dimension 2

Step 2 – Linear Projections (Q, K, V) 🔑

We start from the sequence of token embedding $X \in R^{(d\_model)}$

Compute 3 projections:

$$Q = X \cdot W\_Q$$

$$K = X \cdot W\_K$$

$$V = X \cdot W\_V$$

Dimensions: $W\_Q, W\_K, W\_V \in R^{(d\_model \times d\_k)}$

- $d\_k = d\_model / n\_h$

Query, Key, and Value Intuition 🔑

Each projection has its own learned weight matrix W_{Q,K,V}

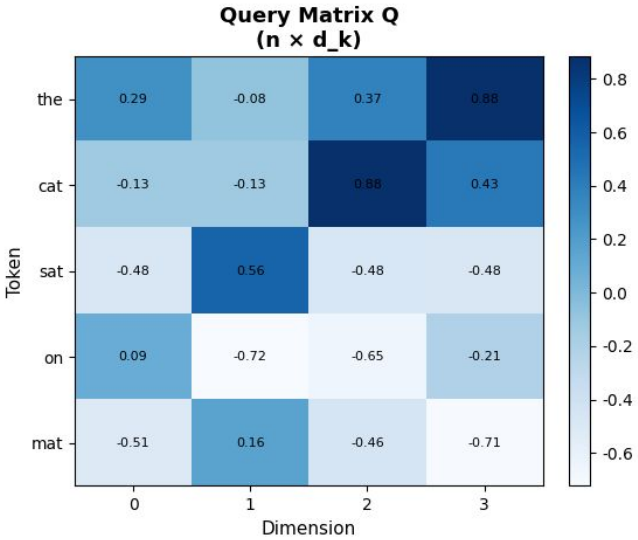
Each view plays a different role:

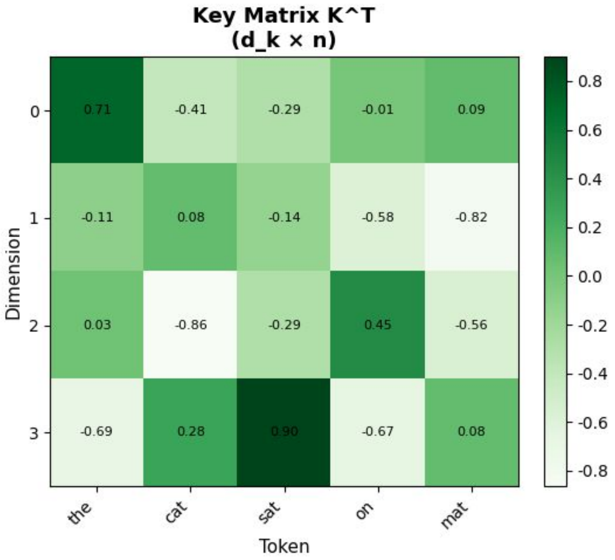
- Query (Q): what this token is trying to find in others
- Key (K): how this token can be found by others
- Value (V): the information this token wants to share

Analogy: Every word asks questions (Q), offers clues (K), and shares information (V) 💡

After this step, each token has its own Q, K, and V vectors

Step 3 – Compute Similarity Scores 📈



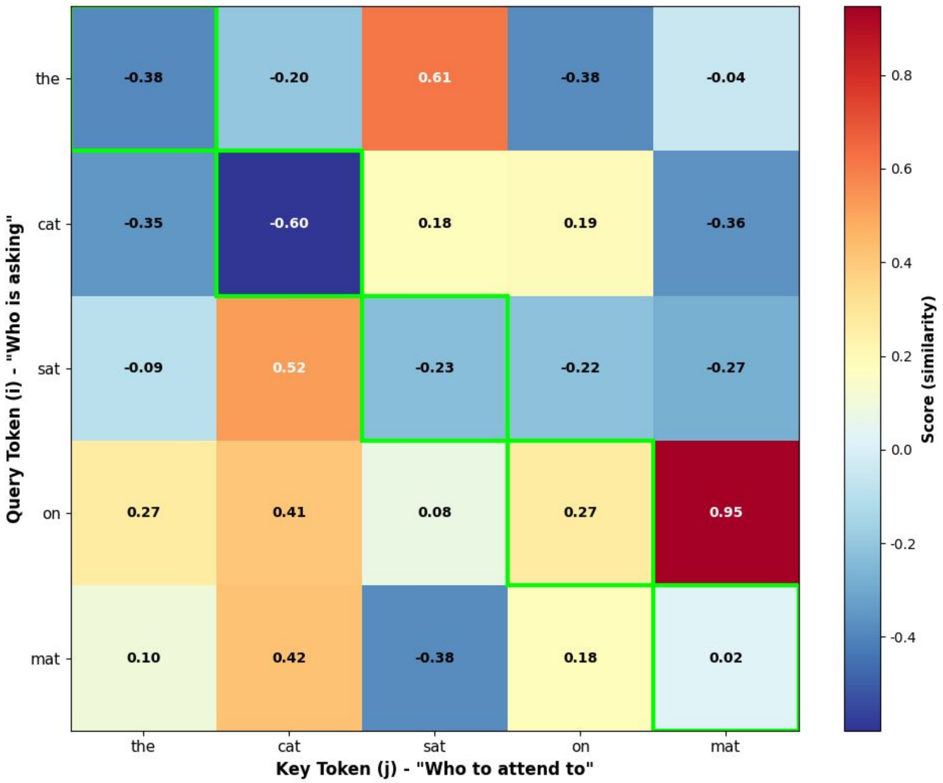


Matrix Multiplication

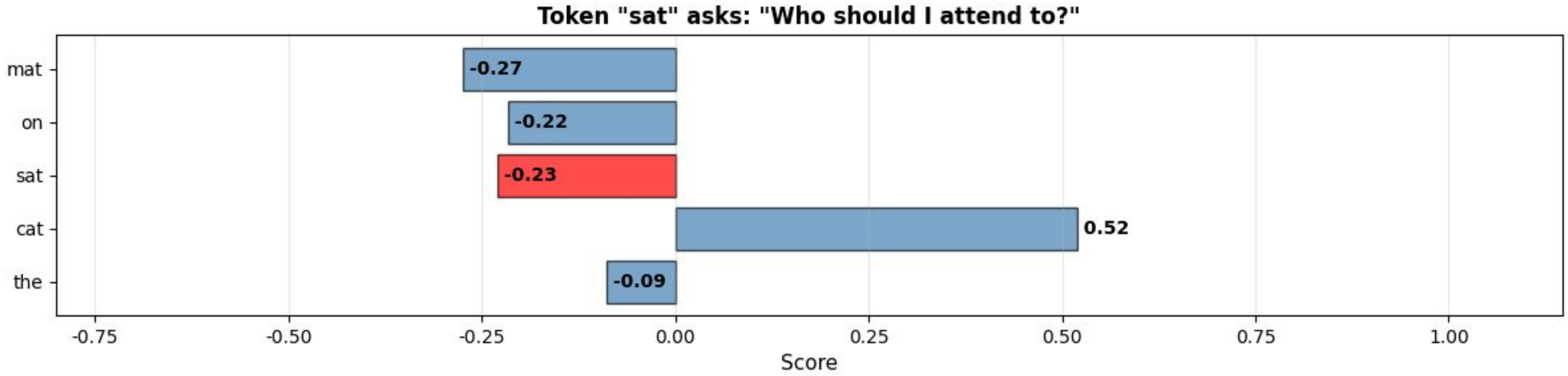
`Q @ K^T`

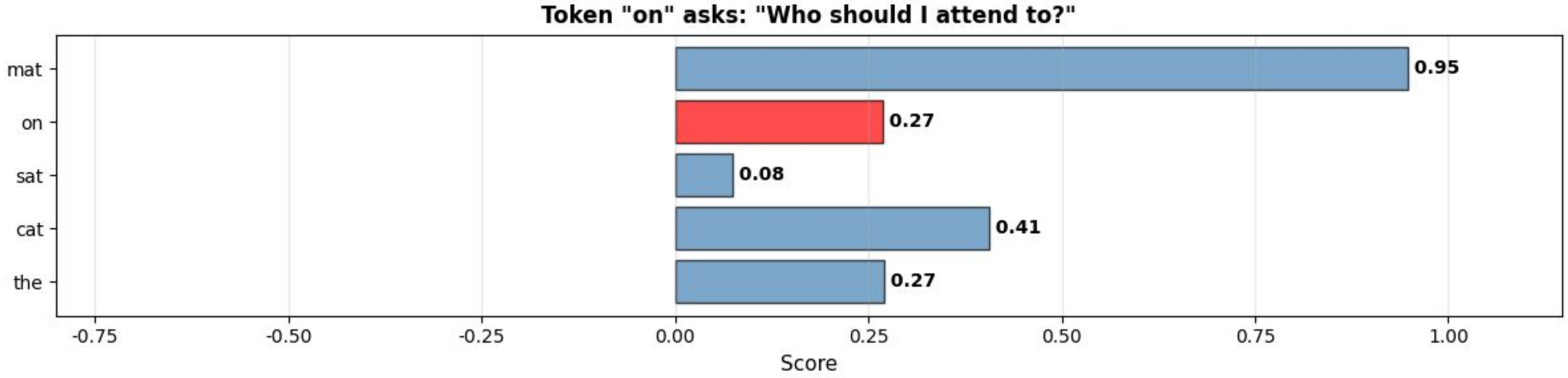
(5 × 4) @ (4 × 5)
= (5 × 5)

Attention Score



Attention Score





Step 4 – Apply Causal Mask ⛓️

In next-token prediction, the model must not peek at future words - it can only use what it has seen so far.

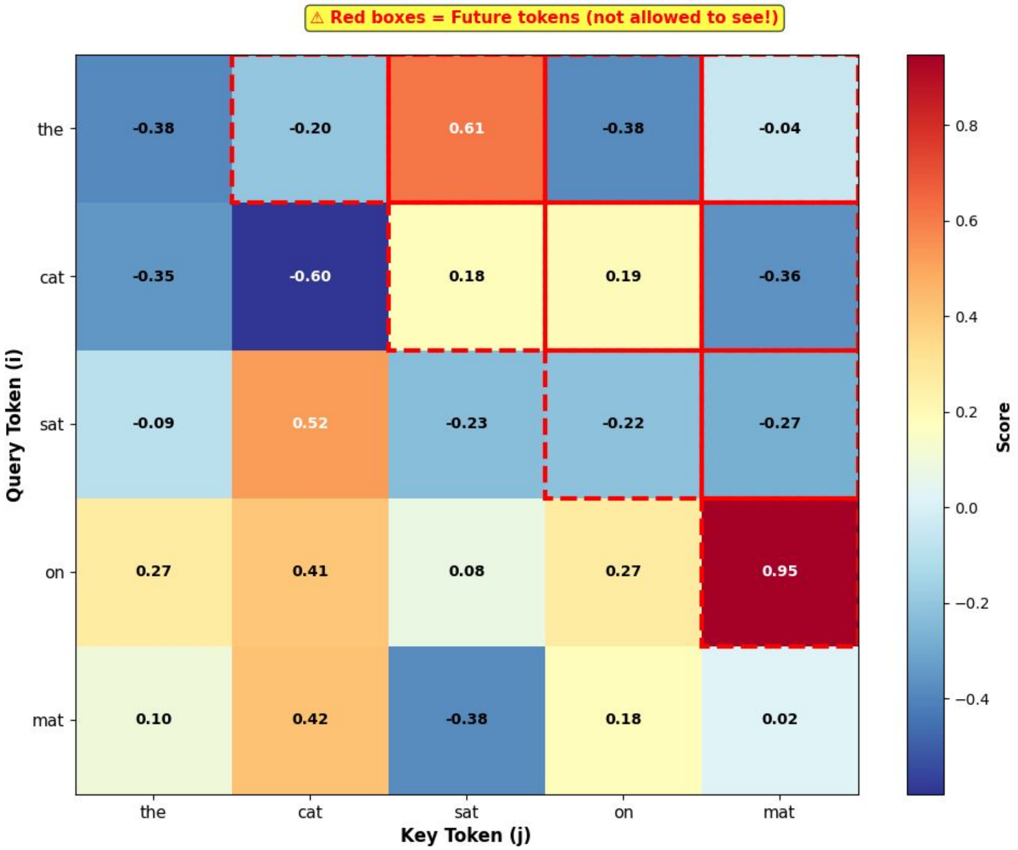
To enforce this rule, we use a causal mask:

- Apply upper-triangular matrix → set to $-\infty$ before softmax

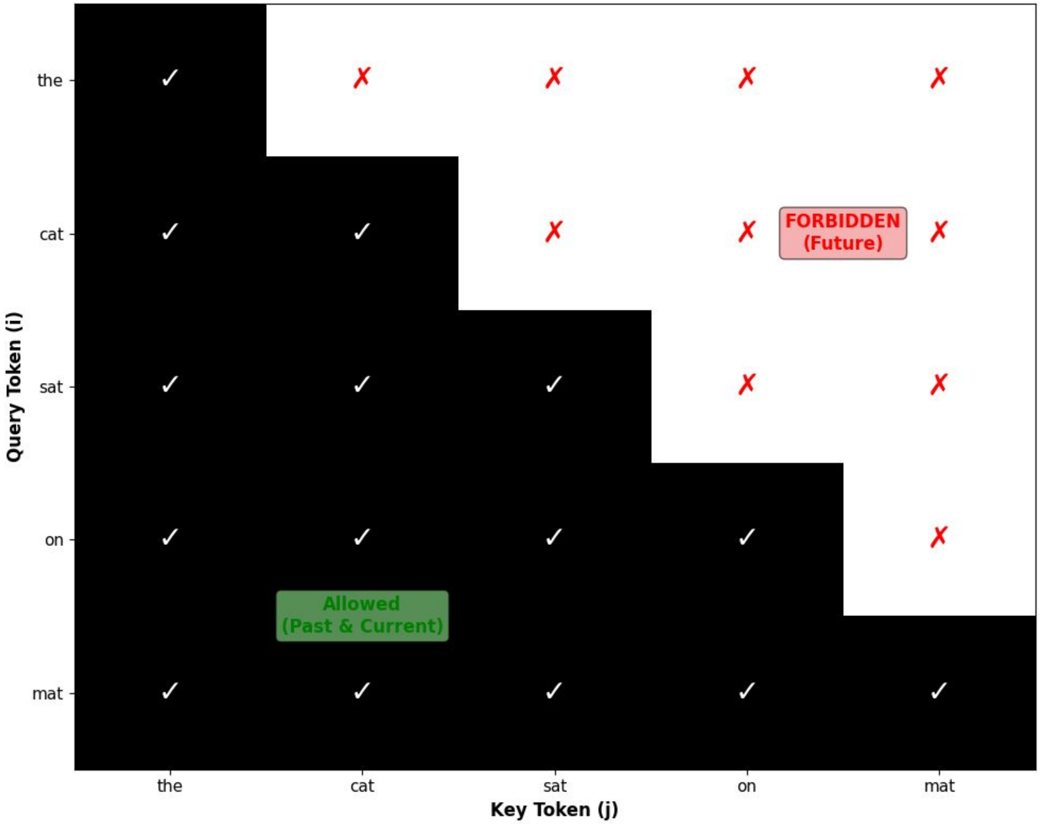
This guarantees autoregressive generation - each token learns only from tokens up to that point, never ahead

At step t, the model can listen only to tokens 1 … t - but not beyond!

Before Masking



Causal Mask



the
cat
sat
on
mat
Query Token (i)
the
cat
sat
on
mat
Key Token (j)
✓
✗
FORBIDDEN
(Future)
Allowed
(Past & Current)

Causal Masking – Which Triangle 🤔

Keep Mask (Allowed)

- Lower Triangle - tokens of positions ≤ i (past + current)

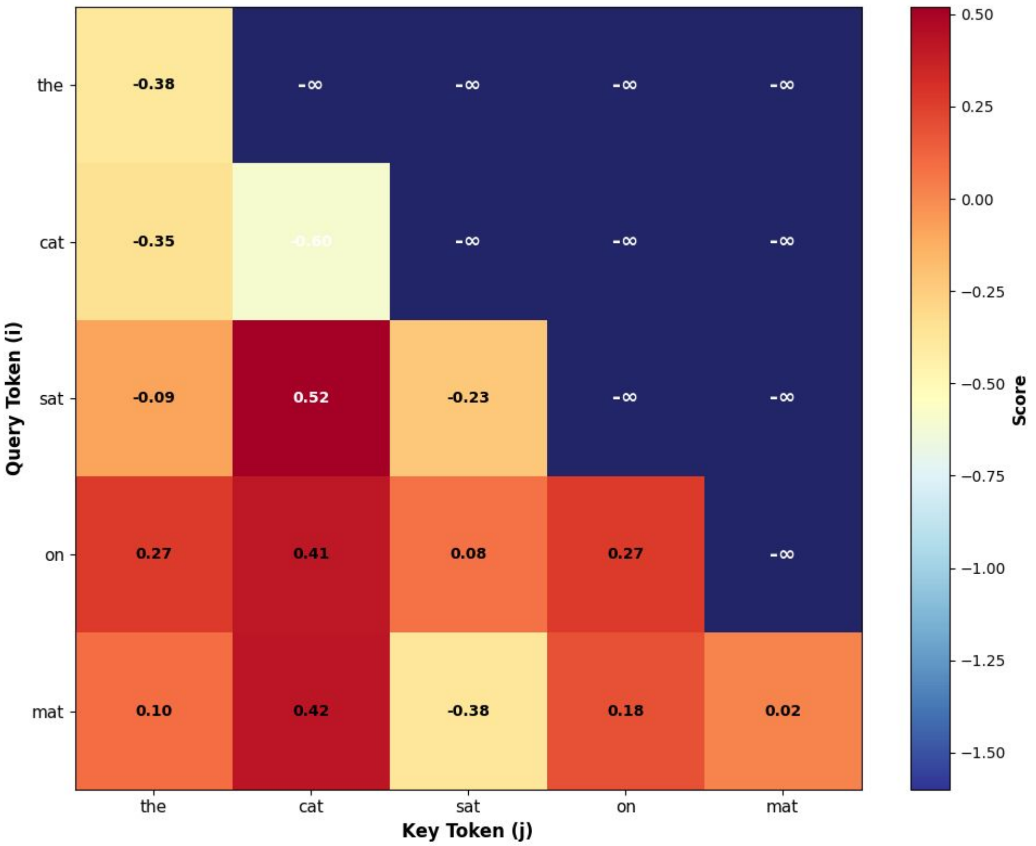
Blocked Mask (Forbidden)

- Upper Triangle -tokens of positions > i (future)

👀 PyTorch Implementation:

```
tgt_mask = torch.triu(torch.ones(T, T) * float('-inf'), diagonal=1)
```

Masked Scores



Step 5 – Normalize with Softmax 🌡️

Convert the similarity scores into probabilities:

$$\text{Weights} = \text{softmax}(\text{Scores} / \text{sqrt}(d\_k))$$

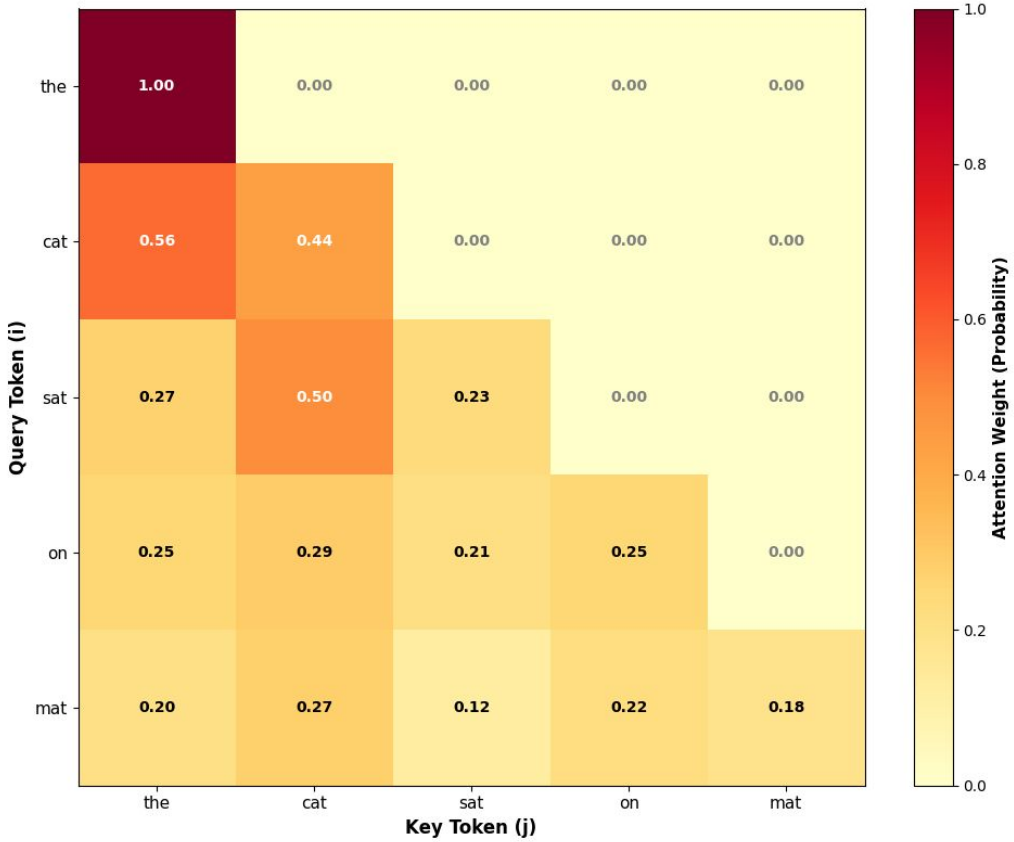
Each row becomes a probability distribution over the visible tokens

The scaling term sqrt(d_k) keeps large values from dominating

The weights always sum to 1, making them easy to interpret as

- prob
- attention focus

Softmax



Query Token (i)
Key Token (j)
the
cat
sat
on
mat
the
cat
sat
on
mat
1.00
0.00
0.00
0.00
0.00
0.56
0.44
0.00
0.00
0.00
0.27
0.50
0.23
0.00
0.00
0.25
0.29
0.21
0.25
0.00
0.20
0.27
0.12
0.22
0.18
Attention Weight (Probability)
0.0
0.2
0.4
0.6
0.8
1.0

Step 6 – Weighted Sum of Values 📦

$$\text{Output} = \text{Weights} \cdot V$$

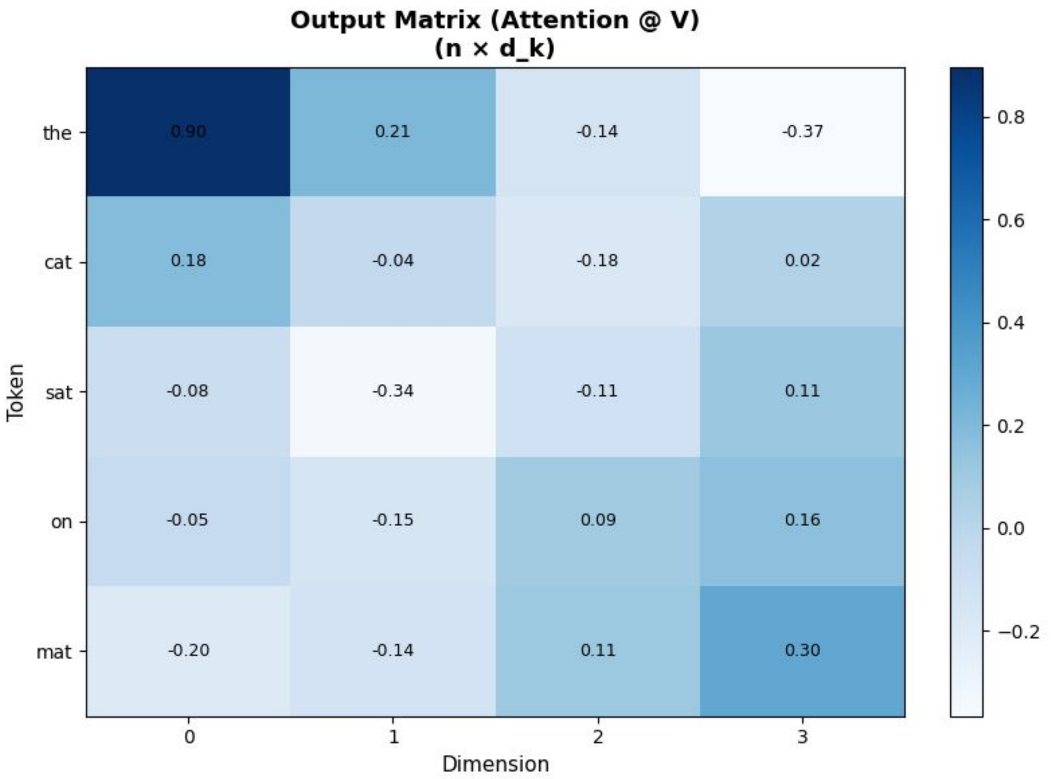
Each token representation = contextual blend of other tokens

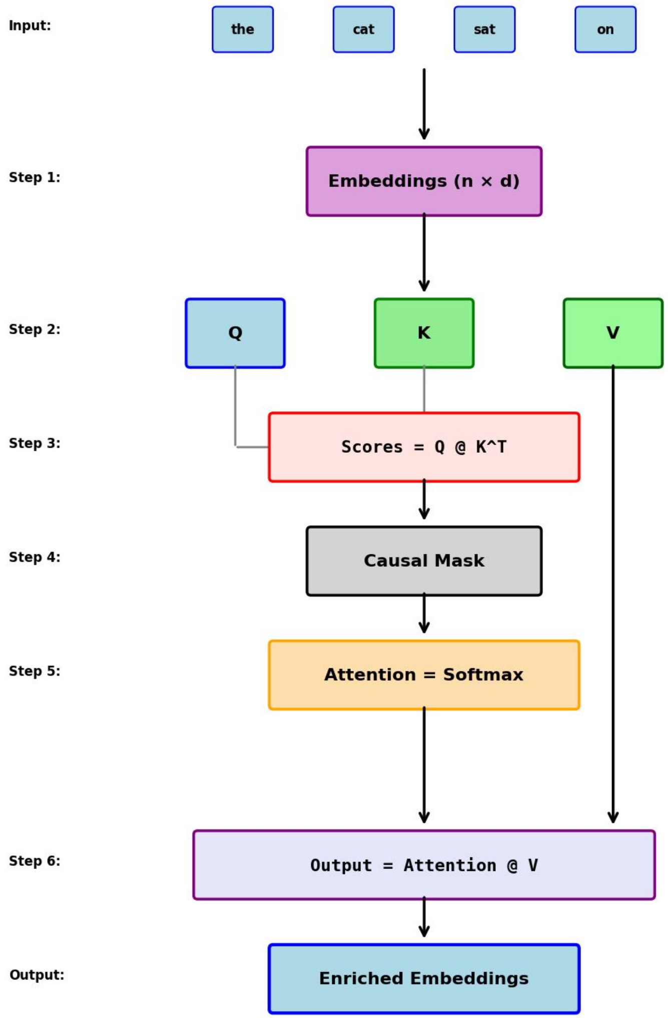
Captures both short- and long-range dependencies

Fully parallelized across the sequence

→ Unlike LSTMs, all tokens are processed simultaneously - faster training and inference

Step 6 – Weighted Sum of Values 📦





🤖 Attention is All You Need

Residual Connections

Why Deep Models Struggle 🤯

As networks grow deeper, information and gradients can vanish or explode

→ **Early layers** might stop learning, or updates can become unstable

When signals vanish or explode, the model forgets what earlier layers learned, slowing convergence or breaking training

We need a way to keep useful information flowing through all layers

→ this motivates the idea of residual/skip connections

The Shortcut Idea ⚡

The layer learns only the difference (the residual) between input and output:

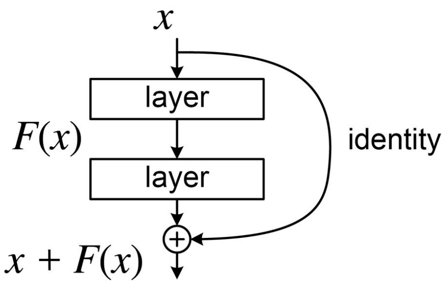
$$F(x) = (x + F(x)) - x$$

This means the model keeps what it already knows and adds new information

→ Preserves useful signals from earlier layers → prevents “forgetting”

Helps keep features alive and makes training more stable

→ Gradients flow CAN directly through the shortcut path



Residuals in Transformers 🤖

Each major sublayer (e.g., Self-Attention, Feed-Forward) has its own shortcut

The model can refine representations without forgetting earlier knowledge

→ Each layer adds subtle improvements instead of relearning everything

This design allows stacking many layers (tens or hundreds!) while keeping training stable and efficient

→ Residuals + normalization prevent gradient vanishing even in deep networks

In practice, this means deep NN can grow very large - yet still converge reliably

Transformer Block

Putting the Pieces Together 🧠

We now have 3 key components:

1️⃣ Self-Attention → lets tokens share information

2️⃣ Residual Connection → keeps original signal flowing

3️⃣ Feed-Forward Network → add non-linearity

Combine them → a single Transformer Block, the building unit of 🤖

Full Transformer Block Flow 🚀

1️⃣ Input sequence → Self-Attention → output context

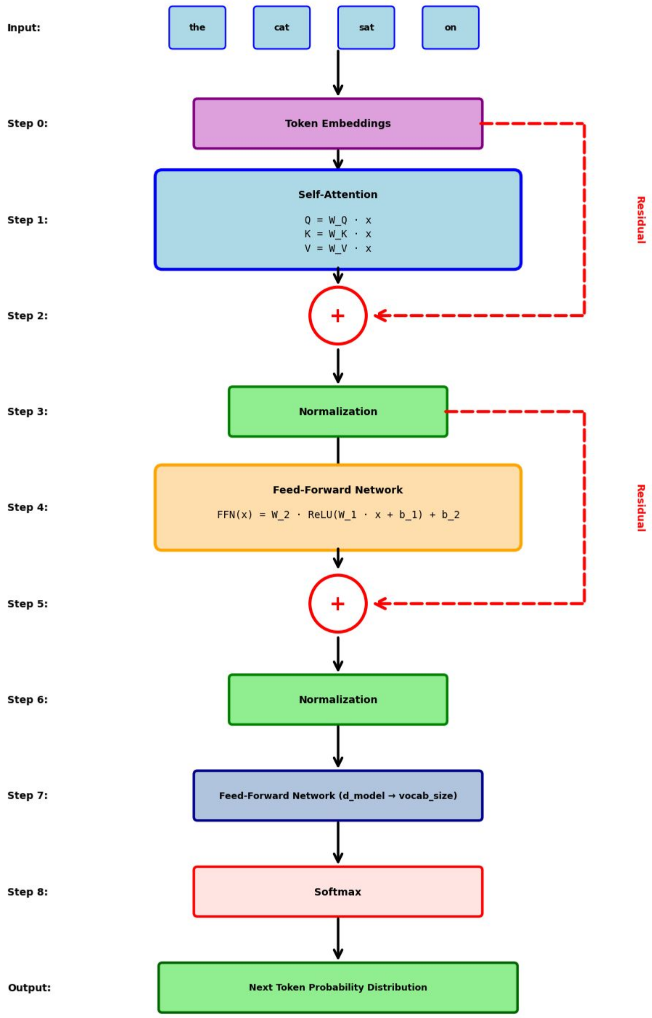
2️⃣ Add a Residual Connection

3️⃣ Output passes into Feed-Forward Network

4️⃣ Add another Residual Connection

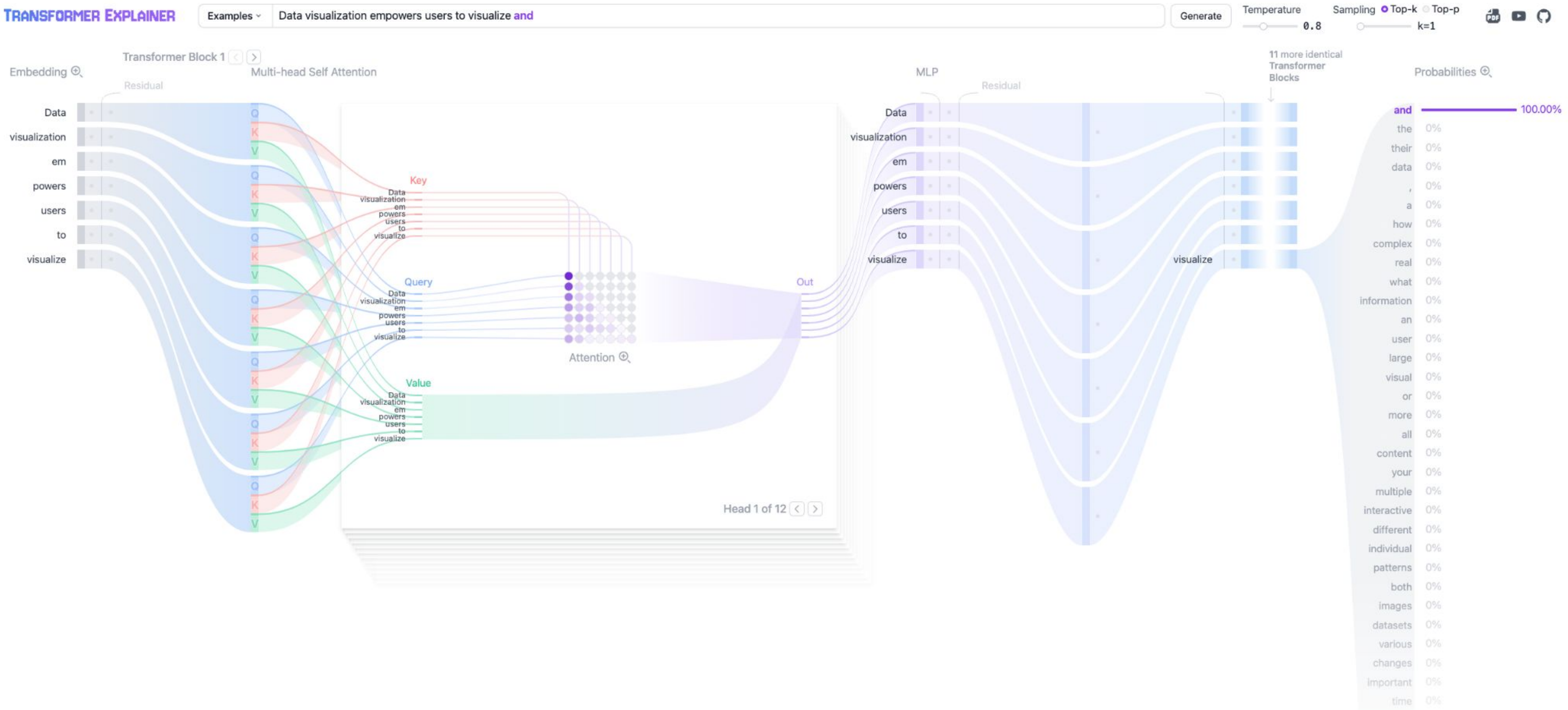
→ Result: new, context-aware token representations

→ This block can be stacked many times for deeper understanding.


Input:
the
cat
sat
on
Step 0:
Token Embeddings
Step 1:
Self-Attention
Q = W_Q · x
K = W_K · x
V = W_V · x
Step 2:
+
Residual
Step 3:
Normalization
Step 4:
Feed-Forward Network
FFN(x) = W_2 · ReLU(W_1 · x + b_1) + b_2
Residual
Step 5:
+
Step 6:
Normalization
Step 7:
Feed-Forward Network (d_model → vocab_size)
Step 8:
Softmax
Output:
Next Token Probability Distribution

Visualize Transformer 🤖

https://poloclub.github.io/transformer-explaine



Summary

Next-Token Prediction: Predicts the next token from previous ones

Self-Attention: Shares context among tokens

Residual Connection: Keeps information flowing, stabilizes training

Feed-Forward Network: Enhances non-linearity to each token

Transformer Block: Combines all 3 - The core unit of modern LLMs