Week 03 & 04: From Text to Attention

The Complete Journey: Tokenization → Embeddings → Positional Encoding → Attention

WEEK 03 & 04 TOPIC NLP FOUNDATIONS DATE 07 FEB 2026

INSTRUCTOR 100XSCHOOL

How does a computer understand "The cat sat on the mat"? This week, we trace the complete journey from raw text to contextualized understanding — exploring how transformers actually "see" language.

Table of Contents

1 Tokenization: Breaking Text into Pieces

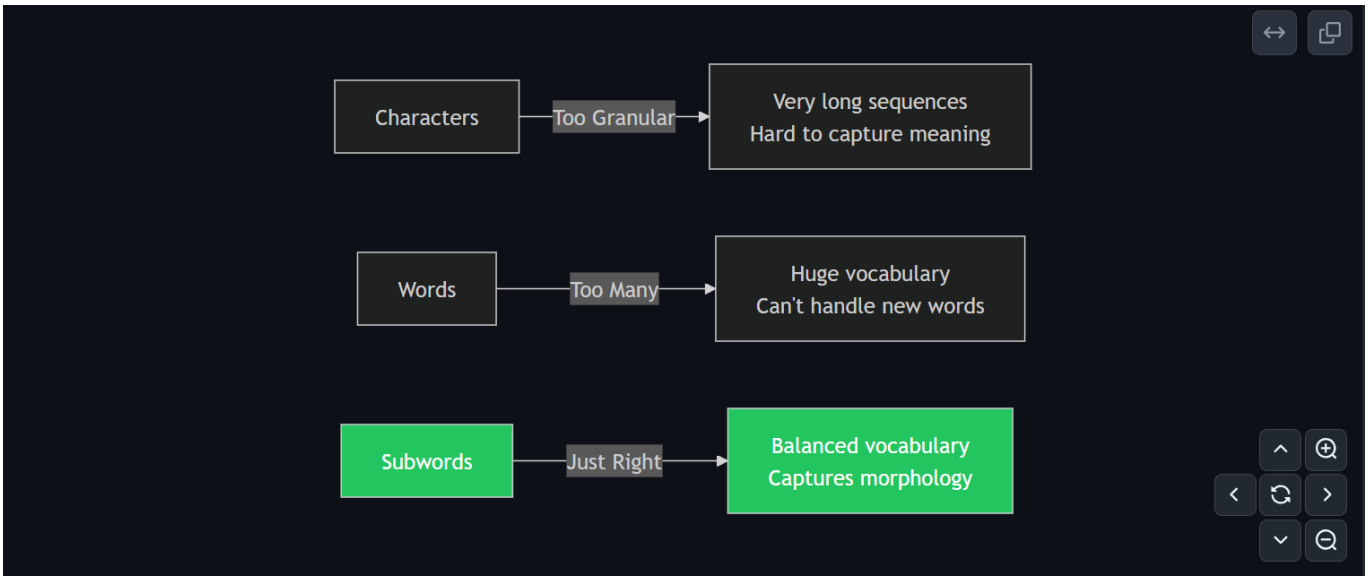
The Model's "Eyes"

> **Tokenization is the foundation of transformer processing.** The tokenizer determines the fundamental units the model uses for "thought."

Neural networks only understand numbers. Tokenization is the crucial first step that transforms human-readable text into numerical tokens — and **how** we tokenize profoundly affects everything downstream.

The Granularity Balance

Three Approaches — Finding "Just Right"



Character-Level Tokenization

Idea: Each character becomes a token.

```
"Hello" → ['H', 'e', 'l', 'l', 'o'] → [72, 101, 108, 108, 111]
```

Pros	Cons
☑ Tiny vocabulary (~256 characters)	❌ Very long sequences
☑ Can handle any text	❌ Hard to capture meaning
☑ No "unknown" tokens	❌ Slow processing

Verdict: ❌ **Too Granular** — sequences become too long and meaning is scattered.

Word-Level Tokenization

Idea: Each word becomes a token.

```
"Hello world" → ['Hello', 'world'] → [1234, 5678]
```

Pros	Cons
☑ Words have clear meaning	❌ Huge vocabulary (millions of words)
☑ Short sequences	❌ Can't handle new words ("COVID-19")
☑ Intuitive	❌ Different forms = different tokens ("run", "running", "ran")

Verdict: ❌ **Too Many** — vocabulary explodes, new words become "unknown."

Subword Tokenization (BPE) ⭐ The Winner

Idea: Break words into meaningful pieces. This is what modern LLMs use!

```
"unhappiness" → ['un', 'happiness'] or ['un', 'happ', 'iness']
```

Pros	Cons
☑ Balanced vocabulary size (~50K-100K)	❌ Token boundaries may seem odd
☑ Handles new words gracefully	❌ Slightly more complex
☑ Captures morphology (prefixes, suffixes)	

Verdict: ☑ **Just Right** — balanced vocabulary that captures meaning and handles novelty.

> [!TIP] **GPT-4 uses `cl100k_base` encoding with a vocabulary of ~100,000 tokens**

🎯 Why Tokenization Matters

> **Tokenization isn't just preprocessing — it has profound practical implications for how models "see" text.**

1️⃣ Context Limits (Operational Impact)

GPT-4's **128K limit** refers to **tokens, not words**. In English, this is roughly 1.3 tokens per word, but code can be much higher.

Content Type	Tokens per Word	128K Tokens ≈
English prose	~1.3	~98K words
Code (Python)	~2.5	~51K words
JSON data	~3+	~40K words
Non-English	~2-4	~32-64K words

> [!WARNING] **Code and structured data consume tokens much faster than natural language!**

2️⃣ Financial Cost

API billing is calculated based on **token counts**, not character counts. Efficient prompting requires understanding token density.

```
import tiktoken
tokenizer = tiktoken.get_encoding("cl100k_base")

# Same meaning, different token costs!
```

```
prompt1 = "Summarize this"       # 2 tokens
prompt2 = "Please provide a comprehensive summary of the following"  # 8 tokens

# 4x the cost for essentially the same instruction!
```

> [!TIP] **Cost optimization:** Keep prompts concise. Avoid verbose instructions when shorter ones work.

3️⃣ Logical Failures (Model Behavior)

Strange failures in **counting letters** or **reversing words** often stem from the model seeing subword units rather than individual characters.

```
# The infamous strawberry problem
text = "strawberry"
tokens = tokenizer.encode(text)
for token in tokens:
    print(f"  '{tokenizer.decode([token])}'")
```

```
  'st'
  'raw'
  'berry'
```

The model sees `['st', 'raw', 'berry']` — it **never sees individual 'r' characters!**

This explains why LLMs struggle with:

- ❌ Counting letters in words
- ❌ Reversing words
- ❌ Spelling tasks
- ❌ Character-level manipulations

4️⃣ Language Equity

Different languages require different numbers of tokens to express the same concept, affecting both **cost and performance** globally.

```
concepts = [
    ("Hello", "English"),
    ("こんにちは", "Japanese"),
    ("مرحبا", "Arabic"),
    ("Здравствуйте", "Russian"),
]

for text, lang in concepts:
```

```
    tokens = tokenizer.encode(text)
    print(f"{lang}: '{text}' → {len(tokens)} tokens")
```

```
English: 'Hello' → 1 token
Japanese: 'こんにちは' → 3 tokens
Arabic: '3 → 'مرحبا tokens
Russian: 'Здравствуйте' → 4 tokens
```

> [!CAUTION] **Non-English languages often require 2-4x more tokens**, making API calls more expensive and reducing effective context length.

Token Efficiency by Content Type

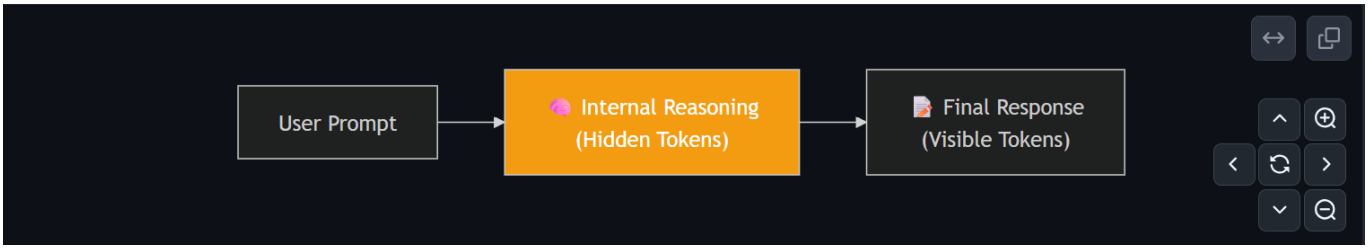
Content Type	Characters	Tokens	Efficiency (chars/token)	Rating
English Prose	44	10	**4.4**	Most Efficient
URLs	48	11	**4.4**	Most Efficient
Python Code	39	11	**3.5**	Moderate
JSON	43	19	**2.3**	Less Efficient
Numbers	32	14	**2.3**	Less Efficient

> [!TIP]
>
> - **English prose is most efficient** — the tokenizer was optimized for natural language
> - **JSON is nearly 2× less efficient** — quotes, colons, and brackets add up!
> - **Numbers are expensive** — each digit sequence often becomes its own token

---

Thinking Models: Extended Reasoning Through Tokens

"Thinking models" like OpenAI's o1 series and Claude's extended thinking mode generate **internal reasoning tokens before producing a response**.



Standard Model	Thinking Model
Prompt → Response	Prompt → *Internal Reasoning* → Response
~500 output tokens	~5,000+ internal tokens + ~500 output tokens

Standard Model	Thinking Model
Fast, single-pass	Slower, multi-step reasoning

Why This Matters for Tokenization

Token Cost Explosion:

- A simple math problem might use **10,000+ internal tokens**
- These tokens aren't shown but ARE processed and billed
- Understanding tokenization helps you predict costs

> [!IMPORTANT] **More tokens = more "thinking time" = better reasoning.** But also significantly more expensive!

🧑‍💻 Practical Tokenization with `tiktoken`

```
import tiktoken

# Load GPT-4's tokenizer
tokenizer = tiktoken.get_encoding("cl100k_base")

# Tokenize text
text = "Hello world"
tokens = tokenizer.encode(text)
print(f"Text: {text}")
print(f"Tokens: {tokens}")
print(f"Token count: {len(tokens)}")
```

Output:

```
Text: Hello world
Tokens: [9906, 1917]
Token count: 2
```

🔍 Understanding Token Boundaries

Where Do Tokens Start and End?

```
text = "Hello world"
tokens = tokenizer.encode(text)

print("Token breakdown:")
for i, token in enumerate(tokens):
```

```
    decoded = tokenizer.decode([token])
    print(f"  Token {i}: {token} → '{decoded}'")
```

Output:

```
Token breakdown:
  Token 0: 9906 → 'Hello'
  Token 1: 1917 → ' world'
```

> [!IMPORTANT] **Notice the space is attached to "world", not "Hello"!** BPE typically treats `" word"` (space+word) as a single token for common words.

Tokenization Experiments

Interesting Tokenization Cases:

```
examples = [
    "hello",           # Simple word
    " hello",          # With leading space
    "Hello",           # Capitalized
    "HELLO",           # All caps
    "don't",           # Contraction
    "3.14159",         # Number
    "🎉",              # Emoji
    "café",            # Accented character
]

for text in examples:
    tokens = tokenizer.encode(text)
    print(f"'{text}' → {tokens} ({len(tokens)} tokens)")
```

Output:

```
'hello' → [15339] (1 token)
' hello' → [24748] (1 token)
'Hello' → [9906] (1 token)
'HELLO' → [13909] (1 token)
'don't' → [3055, 956] (2 tokens)
'3.14159' → [18, 13, 9335, 2946] (4 tokens)
'🎉' → [9468, 234] (2 tokens)
'café' → [66, 5765] (2 tokens)
```

> [!NOTE]
>
> - Different cases = different tokens ("hello" ≠ "Hello" ≠ "HELLO")

> - Contractions get split
> - Numbers use many tokens (expensive!)
> - Emojis need multiple tokens

🧠 Summary: The Model's "Eyes"

Aspect	Key Insight
Granularity	Subwords strike the balance between vocabulary size and sequence length
Operational Impact	Poor tokenization makes the model work harder to understand context
Perception	If the model can't "see" a pattern in the tokens, it can't learn the underlying meaning
Financial	Token count = API cost
Context Limits	128K tokens ≠ 128K words
Language Equity	Non-English languages are disadvantaged by token efficiency

> [!IMPORTANT] **Tokenization determines the fundamental units the model uses for "thought."** Understanding token boundaries explains many LLM limitations.

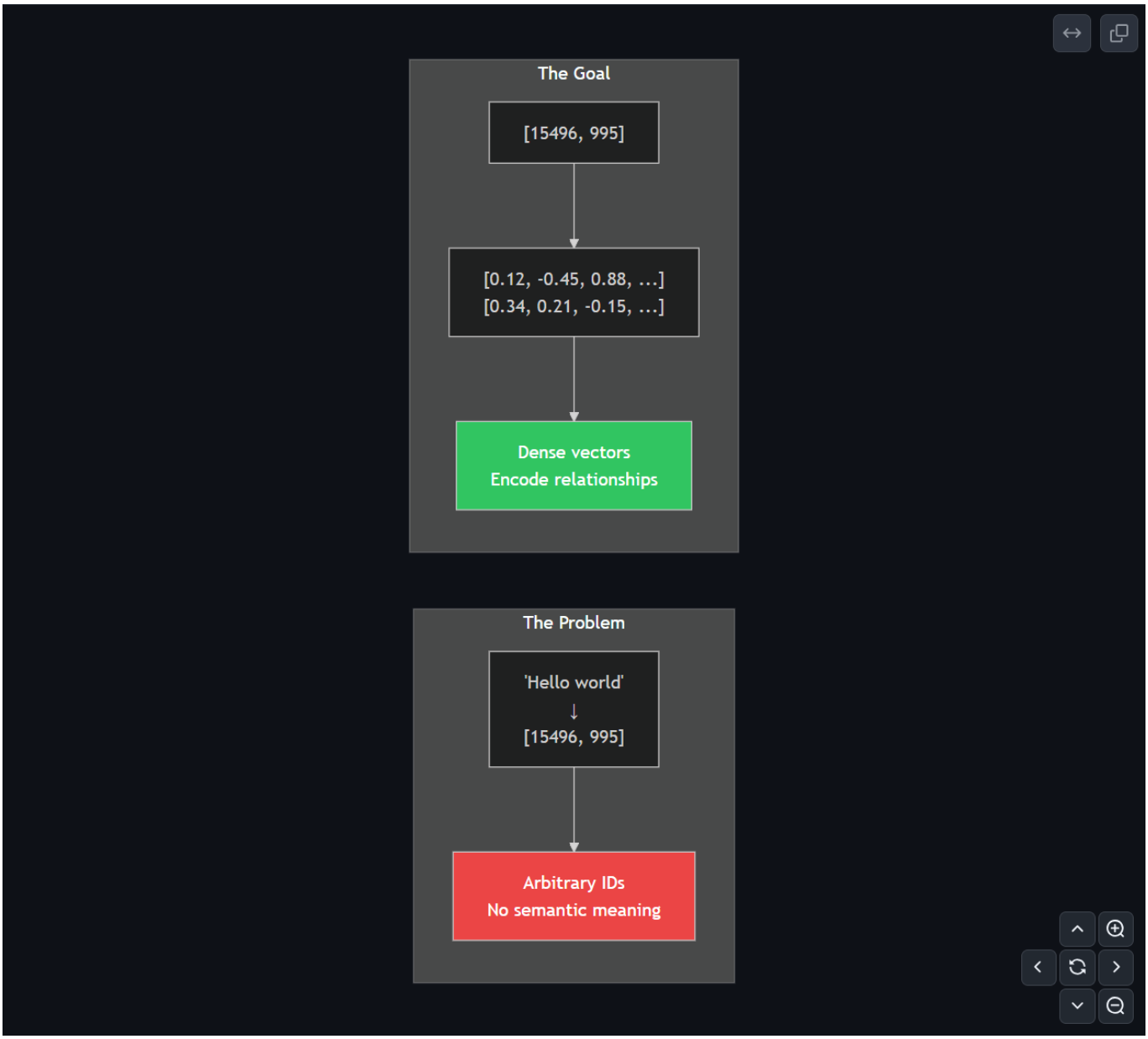
---

2 Embeddings: Meaning in Vectors

From Arbitrary IDs to Meaning

> **The Problem:** After tokenization, we have token IDs like `[15496, 995]`. But these numbers are **arbitrary**. The number 15496 doesn't "know" it represents a greeting, and 995 doesn't "mean" anything about the Earth.

> **The Goal:** We need representations that capture **MEANING**.



We want to transform these discrete, arbitrary integers into **continuous, dense vectors** where the numbers themselves encode semantic relationships.

In this new space:

- "Hello" and "Hi" should be **mathematically similar**
- "Hello" and "Banana" should be **distant**

The Concept of Embeddings

> **Representing tokens as points in high-dimensional space**

Three Core Properties

1 Spatial Logic

Tokens with similar meanings are placed **near each other** in the vector space. **Proximity equals semantic similarity.**

```
"king" and "queen" → close in space (both royalty)
"king" and "pizza" → far apart (unrelated concepts)
```

2️⃣ Vector Representation

Each token becomes a **list of numbers (a vector)** instead of a single ID, allowing for **mathematical operations on meaning**.

```
# Token ID (discrete, arbitrary)
token_id = 15496  # No mathematical meaning

# Embedding vector (continuous, meaningful)
embedding = [0.12, -0.45, 0.88, 0.23, -0.67, ...]  # 768 dimensions
# Each dimension captures some aspect of meaning
```

3️⃣ Multidimensionality

Modern models use **hundreds or thousands** of dimensions to capture the subtle nuances of human language.

Model	Embedding Dimensions
GPT-2 Small	768
GPT-2 Large	1280
GPT-3	12288
GPT-4	(not disclosed, likely 12K+)

> [!TIP] More dimensions = more nuance captured, but also more memory and computation.

📊 The Embedding Matrix: A Lookup Table

The embedding layer is essentially a giant **lookup table**:

```
ID 15496
   ↓
[0.12, -0.45, 0.88, ...]
```

Each token ID acts as an **index** to a specific row in the embedding matrix. This is a simple, fast lookup operation.

```
import numpy as np

# Simulated embedding matrix
vocab_size = 50000       # Number of tokens
embedding_dim = 768      # GPT-2 style dimensions

# In reality, these are LEARNED during training, not random
embedding_matrix = np.random.randn(vocab_size, embedding_dim) * 0.02

print(f"Embedding matrix shape: {embedding_matrix.shape}")
print(f"Memory: ~{vocab_size * embedding_dim * 4 / 1e6:.1f} MB (float32)")
```

Output:

```
Embedding matrix shape: (50000, 768)
Memory: ~153.6 MB (float32)
```

The Lookup:

```
# Get embedding for a token
token_id = 15496  # Some word like "Hello"
embedding = embedding_matrix[token_id]

print(f"Token ID: {token_id}")
print(f"Embedding shape: {embedding.shape}")
print(f"First 10 dimensions: {embedding[:10].round(3)}")
```

The result is a **768-dimensional vector** that represents the token's core semantic profile.

> [!NOTE] **The "Raw" Starting Point:** This vector is the *isolated representation* of the word. It contains the general meaning of the token *before any context* from the surrounding sentence is applied.

Visualizing Semantic Space

> **How models organize concepts across dimensions**

Clustering

Related concepts (e.g., royalty, gender, animals) naturally **group together** in the high-dimensional vector space.

```
Royalty cluster:  king, queen, prince, princess, throne
Animal cluster:   dog, cat, bird, fish, horse
Food cluster:     apple, bread, pizza, rice, soup
```

Emergent Properties

Dimensions like "gender" or "royalty" **emerge automatically during training** without manual labeling.

The model discovers these abstract concepts by processing billions of words of text!

```
Dimension 237: might encode "formality"
Dimension 451: might encode "animate vs inanimate"
Dimension 89:  might encode "positive vs negative sentiment"
```

Semantic Navigation

The mathematical **"distance" between points** represents the semantic similarity of the tokens.

```python
import gensim.downloader as api

# Load pre-trained word vectors
word_vectors = api.load("glove-wiki-gigaword-100")

# Find similar words
similar = word_vectors.most_similar("king", topn=5)
print("Words similar to 'king':")
for word, score in similar:
    print(f"  {word}: {score:.3f}")
```

```
Words similar to 'king':
  prince: 0.824
  queen: 0.768
  throne: 0.763
  monarch: 0.755
  kingdom: 0.742
```

Semantic Vector Arithmetic

> **Mathematical logic embedded in language:** `king - man + woman ≈ queen`

One of the most remarkable properties of word embeddings:

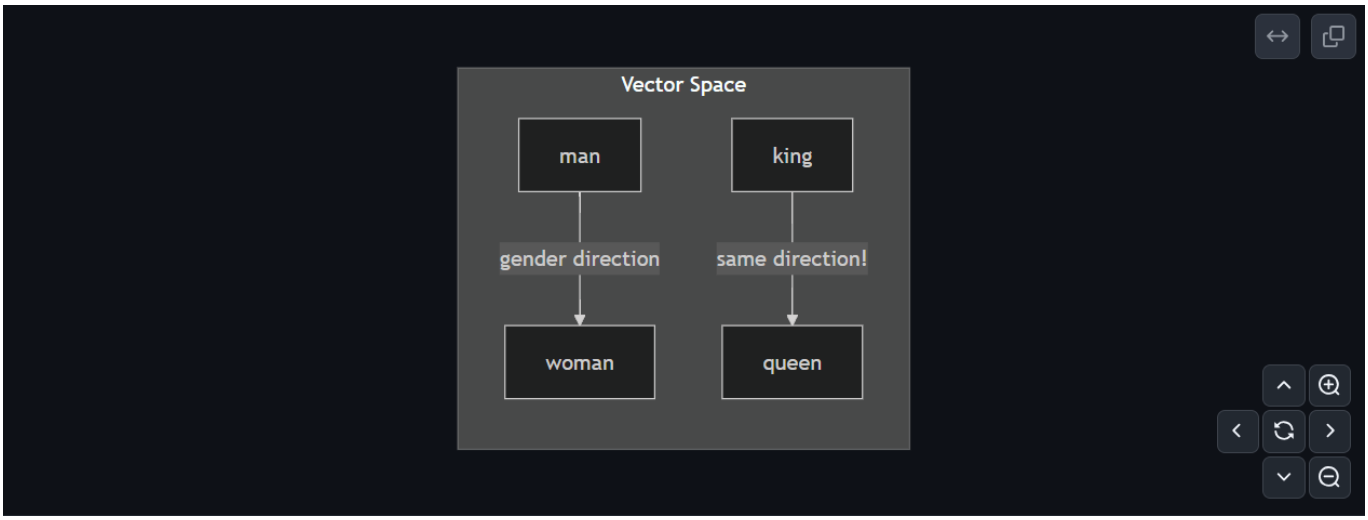
```python
result = word_vectors.most_similar(
    positive=["king", "woman"],
    negative=["man"],
    topn=5
```

```
)

print("king - man + woman = ?")
for word, score in result:
    print(f"  {word}: {score:.3f}")
```

```
king - man + woman = ?
  queen: 0.769
  princess: 0.713
  monarch: 0.694
  empress: 0.683
  prince: 0.671
```

Why This Works: Three Key Insights

Concept	Explanation
Directional Meaning	The man→woman vector mirrors king→queen. Both encode the "gender" direction.
Relational Encoding	Embeddings encode relations as **consistent offsets** in vector space.
Universal Patterns	These offsets recur across large vocabularies — the model learns abstract relationships.



More examples:

```
analogies = [
    (["paris", "germany"], ["france"], "Paris:France :: ?:Germany"),
    (["bigger", "small"], ["big"], "big:bigger :: small:?"),
    (["walking", "swam"], ["walked"], "walked:walking :: swam:?"),
]
```

```
for positive, negative, description in analogies:
    result = word_vectors.most_similar(positive=positive, negative=negative,
topn=1)
    print(f"{description} → {result[0][0]}")
```

```
Paris:France :: ?:Germany → berlin
big:bigger :: small:? → smaller
walked:walking :: swam:? → swimming
```

Visualizing Embedding Space with PCA

Reducing 100D to 2D for Visualization:

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Define semantic groups
word_groups = {
    'Family/Age': ['grandfather', 'man', 'woman', 'adult', 'boy', 'girl', 'child',
'infant'],
}

# Get embeddings and reduce dimensions
words = list(word_groups['Family/Age'])
embeddings = [word_vectors[word] for word in words]

pca = PCA(n_components=2)
embeddings_2d = pca.fit_transform(embeddings)

# Plot
plt.figure(figsize=(10, 8))
for i, word in enumerate(words):
    plt.scatter(embeddings_2d[i, 0], embeddings_2d[i, 1], s=100)
    plt.annotate(word, (embeddings_2d[i, 0] + 0.1, embeddings_2d[i, 1] + 0.1))

plt.xlabel("Gender (roughly)")
plt.ylabel("Age (roughly)")
plt.title("2D Semantic Feature Space Mapping")
plt.show()
```

Result: The 2D projection reveals that:

- **X-axis** roughly encodes gender (male ↔ female)
- **Y-axis** roughly encodes age (young ↔ old)
- Related words cluster together!

> [!TIP] These dimensions **emerged automatically** from training on text — no one explicitly labeled "gender" or "age"!

🧠 Summary: Embeddings

> **Converting discrete tokens into meaningful continuous space**

Core Functions

1. ☑ Transform arbitrary integer IDs into points in a high-dimensional semantic map
2. ☑ Ensure proximity in vector space equals similarity in meaning
3. ☑ Pack multiple layers of semantic nuance into a single dense vector

---

Foundational Layer

Every transformer starts by looking up these pre-trained meanings. It is the **bedrock of all subsequent processing**.

⚠ The Limitation

> [!WARNING] **Embeddings are "bag-of-words" by default.** They represent *isolated meaning* but **ignore the order of words**.
>
> This leads us to our next challenge: **positional encoding**.

---

3 Positional Encoding: Where Are You?

The Problem of Sequence

> **Attention is "bag-of-words" by default**

Order Neutrality

Consider these two sentences:

```
"The dog bit the man"
    vs.
"The man bit the dog"
```

Without extra information, attention treats these two sentences **identically**. The mathematical operations don't care about position.

This is known as **permutation invariance** — the model sees a *set* of words, not a *sequence*.

The Loss of Logic

In human language, **position is often the primary driver of meaning**. Order determines who did what to whom.

- "dog bites man" → Dog is the biter, man is bitten
- "man bites dog" → Man is the biter, dog is bitten

Same words. **Opposite meanings.**

The Requirement

We must find a way to inject **"where"** a word is into **"what"** a word is. We need to make the model aware of the **dimension of time** in language.

```
import tiktoken
tokenizer = tiktoken.get_encoding("cl100k_base")

sentence1 = "dog bites man"
sentence2 = "man bites dog"

tokens1 = set(tokenizer.encode(sentence1))
tokens2 = set(tokenizer.encode(sentence2))

print(f"'{sentence1}' tokens: {tokens1}")
print(f"'{sentence2}' tokens: {tokens2}")
print(f"\nSame set of tokens? {tokens1 == tokens2}")
```

```
'dog bites man' tokens: {18964, 66815, 893}
'man bites dog' tokens: {893, 66815, 18964}

Same set of tokens? True
```

> [!CAUTION] **Without positional encoding, the model cannot distinguish between these completely different sentences!**

Positional Encoding: Adding Sequence Awareness

> **Adding a sense of time and order to the model**

1 Injecting Sequence

Transformers process tokens **in parallel** (not sequentially like RNNs), so we add tags to show **where** each token sits in the sequence.

```
Position 0: "The"   → embedding + position_0_encoding
Position 1: "cat"   → embedding + position_1_encoding
Position 2: "sat"   → embedding + position_2_encoding
```

2️⃣ Vector Addition

A small positional vector is **added** to each embedding, encoding both **meaning** and **position**.

Final Input = Embedding + Positional Encoding

3️⃣ Spatial Awareness

This helps the model tell apart different word orders (e.g., who did what) using position tags.

```
"dog bites man" → [embed_dog + pos_0, embed_bites + pos_1, embed_man + pos_2]
"man bites dog" → [embed_man + pos_0, embed_bites + pos_1, embed_dog + pos_2]
```

Now these are **mathematically different** inputs!

How Positional Patterns Work

> **Using mathematical waves to encode location**

Periodic Functions

We use **sine and cosine** waves of varying frequencies. Each dimension of the embedding follows a different wave.

```
Dimension 0: sin(pos / 10000^(0/d))      → Fast oscillation
Dimension 1: cos(pos / 10000^(0/d))      → Fast oscillation
Dimension 2: sin(pos / 10000^(2/d))      → Slower oscillation
...
Dimension d: sin(pos / 10000^(d/d))      → Very slow oscillation
```

Unique Fingerprint

Every position in the sequence (0, 1, 2, ...) gets a **unique combination of values**, creating a "fingerprint" for that location.

Position	Dim 0	Dim 1	Dim 2	Dim 3	...
0	0.000	1.000	0.000	1.000	...

Position	Dim 0	Dim 1	Dim 2	Dim 3	...
1	0.841	0.540	0.100	0.995	...
2	0.909	-0.416	0.199	0.980	...
3	0.141	-0.990	0.296	0.955	...

Relative Distance

The model can calculate the **relative distance** between words because the waves are mathematically related.

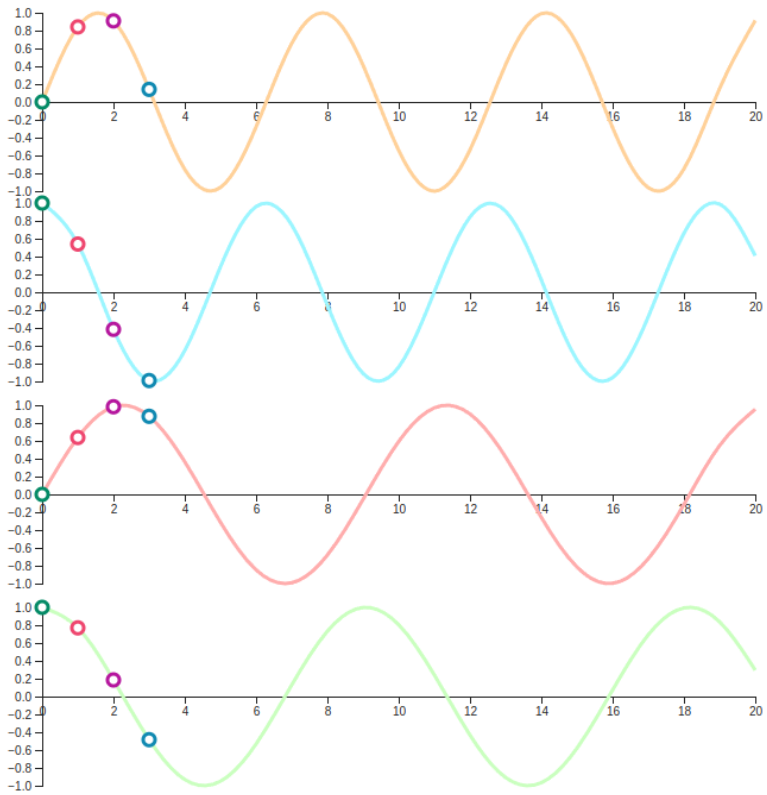
For any fixed offset **k**, the position **p+k** can be represented as a linear function of position **p**. This allows the model to learn to attend by relative position easily.

Generalization

This method allows the model to **handle sequences longer than those it saw during training**. Since the waves are continuous, the model can extrapolate to new positions.

Visual Intuition

Sine Waves at Different Frequencies



p0	p1	p2	p3	
0.000	0.841	0.909	0.141	i=0
1.000	0.540	-0.416	-0.990	i=1
0.000	0.638	0.983	0.875	i=2
1.000	0.770	0.186	-0.484	i=3

Positional Encoding

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Settings: d = 50
The value of each positional encoding depends on the *position* (*pos*) and *dimension* (*d*). We calculate result for every *index* (*i*) to get the whole vector.

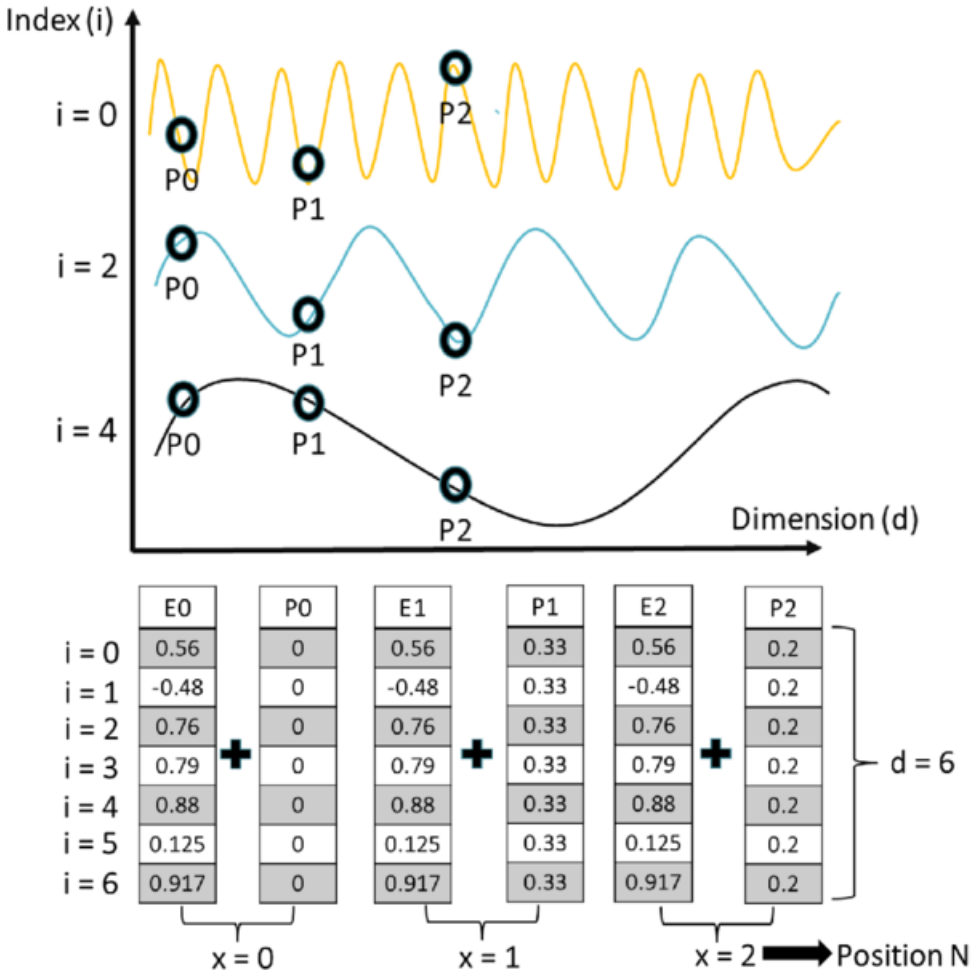
What this shows:

- Each row is a **different dimension** (i=0, 1, 2, 3)
- The waves have **different frequencies** (top = fast, bottom = slow)

- The colored circles show values at positions **p0, p1, p2, p3**

Key observation: At i=0 (fast), values change rapidly. At i=3 (slow), values change gradually.

How Positions Get Their Unique Fingerprints



Each column in the matrices is a **unique positional fingerprint**. Even though each wave is simple, combining many waves at different frequencies creates a **unique pattern for every position!**

🎵 Why Sine Waves?

> **The logic behind the original transformer's choice**

Mathematical Advantages

Property	Benefit
Relative Position	For any fixed offset k, position p+k can be expressed as a linear function of position p
Generalization	Since waves are continuous, the model can extrapolate to **longer sequences** than those seen during training

Property	Benefit
Bounded [-1, 1]	Values stay stable, won't explode during training
Unique per position	Each position has a distinguishable signature

Efficiency & Logic

Property	Benefit
No Learned Parameters	Unlike other methods, sinusoidal encoding doesn't require extra parameters to learn. The patterns are **fixed and pre-calculated**.
Unique Multi-Scale Patterns	By using multiple frequencies, the model creates a **unique fingerprint** for every single position in a very high-dimensional space.

The Formula

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))

PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

Where:

- `pos` = position in the sequence (0, 1, 2, ...)
- `i` = dimension index (0, 1, 2, ... d_model/2)
- `d_model` = embedding dimension

```
import numpy as np

def get_positional_encoding(seq_length, d_model):
    """Generate sinusoidal positional encoding."""
    position = np.arange(seq_length)[:, np.newaxis]    # (seq_length, 1)
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))

    pe = np.zeros((seq_length, d_model))
    pe[:, 0::2] = np.sin(position * div_term)  # Even indices: sine
    pe[:, 1::2] = np.cos(position * div_term)  # Odd indices: cosine

    return pe

# Generate for a sequence of 50 positions, 64 dimensions
pe = get_positional_encoding(50, 64)
print(f"Positional encoding shape: {pe.shape}")
```

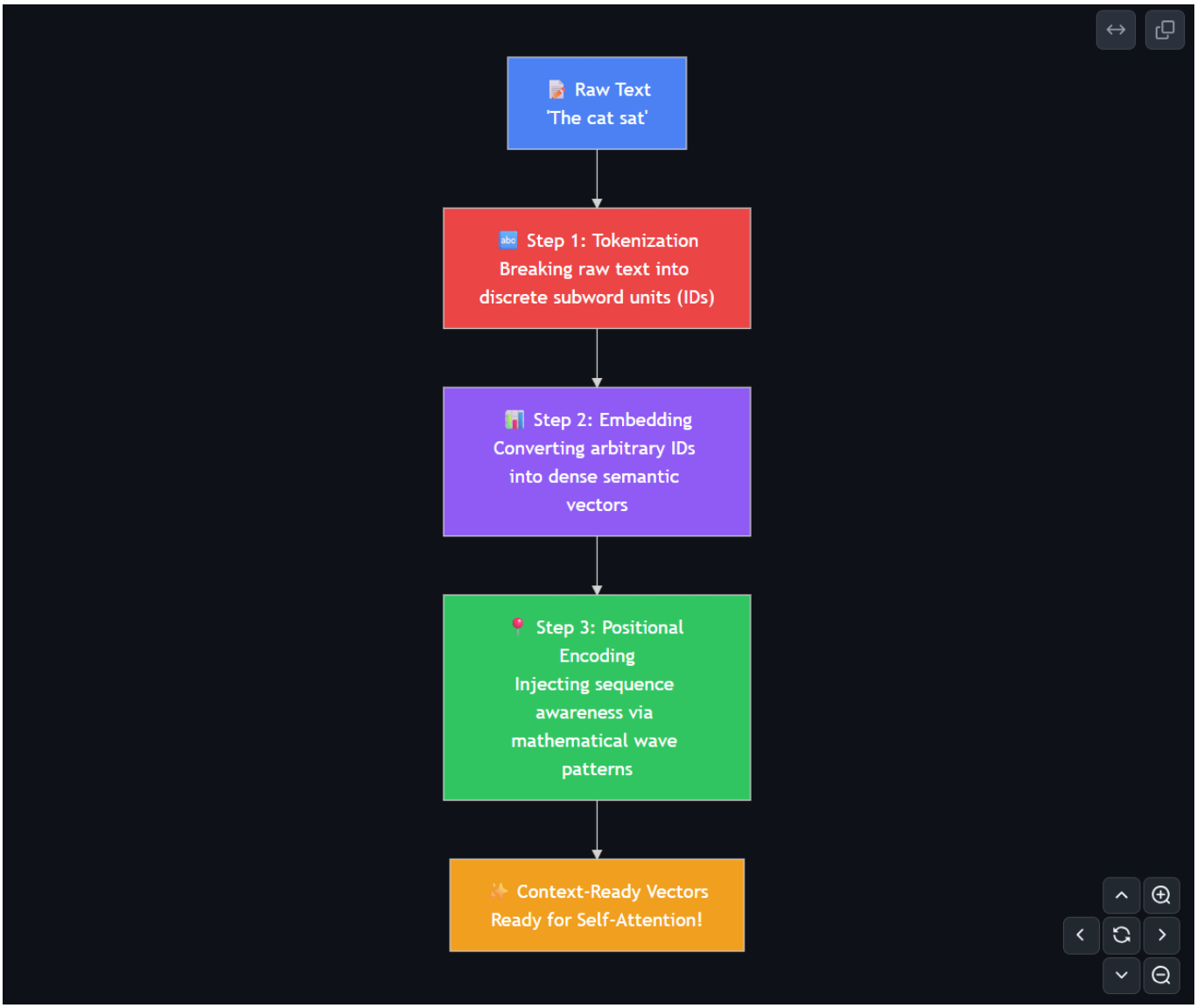
📊 Combining Position with Meaning

```
# Example for "The cat sat"
sentence = ["The", "cat", "sat"]
d_model = 8  # Small for demonstration

# Simulated embeddings (semantic meaning)
np.random.seed(42)
embeddings = np.random.randn(3, d_model) * 0.1

# Get positional encodings (position information)
pe = get_positional_encoding(3, d_model)

# Combine: meaning + position
final_input = embeddings + pe

print("Word embeddings (meaning only):")
print(embeddings.round(3))
print("\nPositional encodings (position only):")
print(pe.round(3))
print("\nFinal input (meaning + position):")
print(final_input.round(3))
```

Now each word's representation contains **BOTH**:

- ☑ Semantic meaning (from embeddings)
- ☑ Position information (from positional encoding)

📋 The Pipeline So Far



Step	What Happens	Output
Tokenization	Breaking raw text into discrete subword units (IDs)	`[464, 3797, 3332]`
Embedding	Converting arbitrary IDs into dense semantic vectors	`[[0.1, -0.2, ...], ...]`
Positional Encoding	Injecting sequence awareness via mathematical wave patterns	Vectors now know their position!

> [!NOTE] Now we're ready for the **core of the transformer**: Self-Attention!

---

4 Self-Attention: The Magic of Transformers

The Core of the Transformer

> **Self-Attention: The defining feature that allows the model to process information *globally* rather than sequentially.**

It is the heart of the "Attention is All You Need" philosophy.

The Engine of Context

Self-Attention is what makes transformers different from older models like RNNs and LSTMs. It processes information **globally** rather than sequentially.

Traditional Models (RNN)	Transformers (Attention)
Process tokens one-by-one	Process all tokens simultaneously
Information passes through a "bottleneck"	Every token sees every other token directly
Long-range dependencies are hard	Long-range dependencies are easy
Sequential (slow)	Parallel (fast)

Mutual Influence

Every token in a sequence **"attends" to every other token simultaneously**, creating a rich, multi-layered map of relationships.

```
"The cat sat on the mat"
     ↓
Every word can see every other word at once!
```

Why It Matters

Benefit	Explanation
Contextualization	Words don't have a single fixed meaning; their meaning is **refined and updated** based on the words surrounding them in the specific sequence.
Interconnectivity	Self-attention builds a **dense web of connections**, allowing the model to resolve ambiguities and understand complex logical dependencies.

🎯 The Intuition of Attention

> **Focusing on what matters for understanding**

Selective Focus

Filtering Noise: Attention allows the model to **ignore irrelevant words** and amplify the signal from the most related tokens in the sequence.

```
"The cat sat on the mat because it was tired"
                              ↑
                    What does "it" refer to?
                    Attention helps focus on "cat"!
```

Dynamic Weighting: The model assigns a unique **"importance score"** to every word pair, determining exactly how much they should influence each other.

Reference Resolution

Solving Ambiguity: In the sentence "The bank was closed because it was a holiday," attention helps the model know that "it" refers to the "bank."

Semantic Context: The meaning of a word is defined by its relationships. Attention is the tool the model uses to **map those relationships**.

Word	Attends To	Why?
"it"	"bank"	To resolve what "it" refers to
"closed"	"bank"	To understand what was closed
"holiday"	"closed"	To understand why it was closed

🔑 The Three Roles of Every Word (Q, K, V)

> **Queries, Keys, and Values — The information retrieval model**

Each word gets transformed into THREE different representations:

Query (Q) — "What am I looking for?"

The **search criteria** used by a token to find relevant information in the sequence.

```
When processing "sat", the Query asks:
"Who sat? What did they sit on? Why?"
```

Key (K) — "What do I contain?"

The **label or index** that other tokens use to determine if this token is relevant to their search.

```
The word "cat" has a Key that says:
"I'm an animal, a subject, something that can perform actions"
```

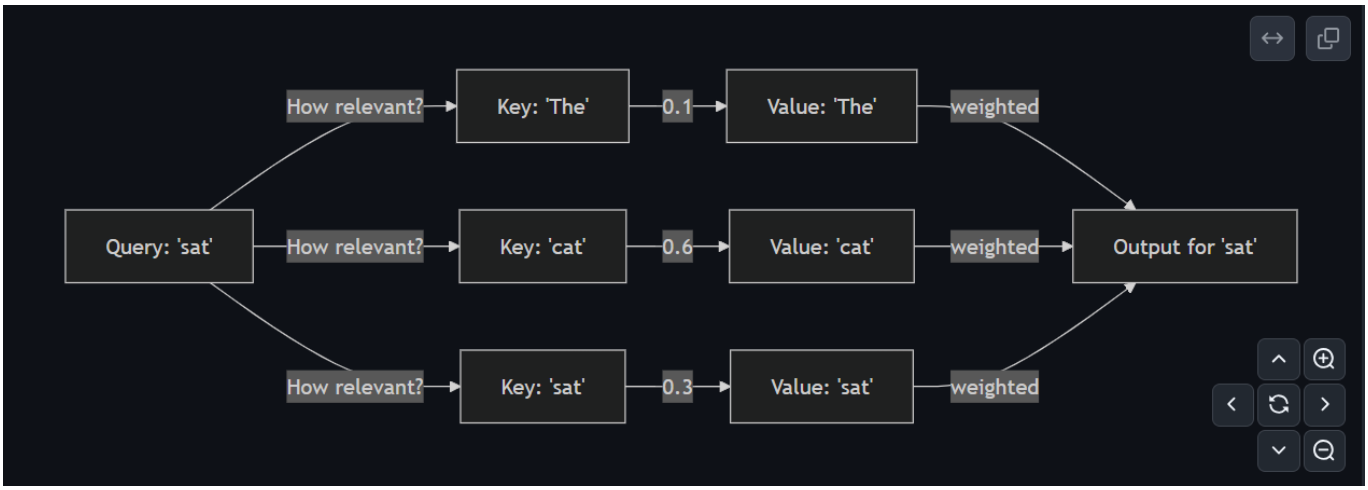
Value (V) — "What information do I offer?"

The **actual semantic content** that is passed through if the Query and Key match.

```
If "sat" finds "cat" relevant, it retrieves the
Value of "cat" — its actual semantic information
```

The Interaction

The **similarity score** between Q and K determines the "attention weight" given to the corresponding Value.



The Database Analogy

> **Understanding attention as a "soft" lookup**

Standard vs. Soft

Hard Lookup (Traditional Database)	Soft Lookup (Attention)
Finds an **exact match** for a key	Finds **partial matches** across all keys
Returns a **single value**	Returns a **mixture of values**
Binary: match or no match	Continuous: degree of match
`SELECT * FROM users WHERE id = 5`	Weighted sum based on similarity

The Mathematics

Component	Role
Weighted Sum	The final representation of a word is a **weighted combination** of all other words in the sequence
Softmax Function	Converts raw similarity scores into **probabilities** that sum to 1, determining the "mix" of values

```
# Hard lookup: only one result
db_result = database.get(key=5)  # Returns exactly one row

# Soft lookup (attention): weighted mixture
attention_result = sum(weight_i * value_i for all i)  # Returns weighted
combination
```

Computing Q, K, and V

> **The mathematical transformation of embeddings**

The Transformation

A single word embedding is multiplied by three separate **learned matrices** (W_Q, W_K, W_V) to create its Query, Key, and Value vectors.

```
# Input embedding: shape (seq_length, d_model)
X = [embedding_0, embedding_1, embedding_2, ...]

# Learned weight matrices: shape (d_model, d_k) or (d_model, d_v)
W_Q = ...  # Learned during training
W_K = ...  # Learned during training
W_V = ...  # Learned during training

# Compute Q, K, V for all words simultaneously
Q = X @ W_Q  # Shape: (seq_length, d_k)
K = X @ W_K  # Shape: (seq_length, d_k)
V = X @ W_V  # Shape: (seq_length, d_v)
```

Learned Behavior

During training, the model learns exactly how to project embeddings to maximize the usefulness of the attention mechanism.

Three Perspectives

Perspective	Role	Description
The Searcher (Q)	Find context	The version of the word used to **find context** in other words
The Content (V)	Provide information	The version of the word that **provides information** once a match is found
The Label (K)	Be found	The version used by **other words to find this one**

> [!TIP] Think of it like a library:
>
> - **Query (Q):** Your search terms
> - **Key (K):** The index/catalog
> - **Value (V):** The actual book content

📐 The Attention Formula

> **Breaking down the famous equation**

Attention(Q, K, V) = Softmax(QK^T / √d_k) × V

This single formula defines the core logic of modern AI, allowing for the **dynamic exchange** of information across a sequence.

The Components

Component	Shape	Role
Dot Product (QK^T)	(seq, seq)	Measures the **similarity** between every Query and every Key in the sequence
Scaling (√d_k)	scalar	Prevents the dot products from becoming too large, ensuring **stable gradients** during training
Softmax	(seq, seq)	Converts raw similarity scores into **attention weights** (probabilities) that sum to 1
Weighted Sum (×V)	(seq, d_v)	Applies the weights to the Values to produce the final **context-aware** representation

---

Step-by-Step Calculation

```
import numpy as np

# Setup
sentence = ["The", "cat", "sat"]
d_model = 8   # Input embedding dimension
d_k = 4       # Query/Key dimension
d_v = 4       # Value dimension

# Random embeddings and weight matrices
np.random.seed(42)
X = np.random.randn(3, d_model)
W_Q = np.random.randn(d_model, d_k) * 0.1
W_K = np.random.randn(d_model, d_k) * 0.1
W_V = np.random.randn(d_model, d_v) * 0.1

# Step 1: Compute Q, K, V
```

```
Q = X @ W_Q  # (3, 4)
K = X @ W_K  # (3, 4)
V = X @ W_V  # (3, 4)

# Step 2: Compute attention scores (QK^T)
scores = Q @ K.T  # (3, 3)

# Step 3: Scale by sqrt(d_k)
d_k_scalar = Q.shape[-1]
scaled_scores = scores / np.sqrt(d_k_scalar)

# Step 4: Apply softmax
def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

attention_weights = softmax(scaled_scores)

# Step 5: Weighted sum of values
output = attention_weights @ V  # (3, 4)

print(f"Attention weights:\n{attention_weights.round(3)}")
print(f"\nOutput shape: {output.shape}")
```

🎨 Visualizing the Attention Matrix

> **Mapping how words connect to each other**

Mapping Connections

The attention matrix is a **square grid** where every word in the sequence is compared to every other word.

```
          The   cat   sat
     ┌─────────────────────┐
The  │ 0.4   0.3   0.3     │  ← How "The" distributes attention
cat  │ 0.2   0.5   0.3     │  ← How "cat" distributes attention
sat  │ 0.1   0.6   0.3     │  ← How "sat" distributes attention
     └─────────────────────┘
           ↑
    How much attention each word receives
```

Heatmap Logic

Each cell represents an attention weight. **Brighter cells indicate a stronger relationship** between the tokens.

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(6, 5))
plt.imshow(attention_weights, cmap='Blues')
plt.colorbar(label='Attention Weight')

plt.xticks(range(len(sentence)), sentence)
plt.yticks(range(len(sentence)), sentence)
plt.xlabel('Attends TO (Keys)')
plt.ylabel('Attends FROM (Queries)')
plt.title('Attention Weights')

for i in range(len(sentence)):
    for j in range(len(sentence)):
        plt.text(j, i, f'{attention_weights[i,j]:.2f}',
                 ha='center', va='center', fontsize=12)
plt.show()
```

Key Patterns to Look For

Pattern	Meaning
Self-Focus	Words often attend strongly to themselves, preserving their own identity while incorporating context
Context-Focus	The model learns to attend to logically related tokens, such as **pronouns to their antecedents** or **verbs to their subjects**
Diagonal dominance	Strong self-attention (each word attending to itself)
Off-diagonal bright spots	Important relationships between different words

---

📤 Computing the Output

Finally, each word's output is a **weighted combination of all Value vectors**:

```
# Output = attention_weights @ V
output = attention_weights @ V  # (3, 3) @ (3, 4) = (3, 4)

print(f"Output shape: {output.shape}  (3 words, each with {d_v}D output)\n")

for i, word in enumerate(sentence):
    print(f"'{word}' output: {output[i].round(3)}")
```

```
Output shape: (3, 4)  (3 words, each with 4D output)

'The' output: [-0.047  0.019  0.212  0.095]
```

```
'cat' output: [-0.017  0.007  0.223  0.073]
'sat' output: [0.023 0.005 0.211 0.041]
```

What happened? Each word's output is now **contextualized** — it contains information from other words, weighted by the attention scores.

- If "cat" attended strongly to "sat", then "cat"'s output contains information from "sat"'s Value vector.
- The word "cat" now "knows about" what it sat, where it sat, etc.

Complete Self-Attention Function

Putting It All Together:

```
def self_attention(X, W_Q, W_K, W_V):
    """
    Compute self-attention.

    Args:
        X: Input embeddings, shape (seq_length, d_model)
        W_Q: Query weight matrix, shape (d_model, d_k)
        W_K: Key weight matrix, shape (d_model, d_k)
        W_V: Value weight matrix, shape (d_model, d_v)

    Returns:
        output: Contextualized embeddings, shape (seq_length, d_v)
        attention_weights: Attention pattern, shape (seq_length, seq_length)
    """
    # Step 1: Compute Q, K, V
    Q = X @ W_Q
    K = X @ W_K
    V = X @ W_V

    # Step 2: Compute attention scores
    d_k = Q.shape[-1]
    scores = Q @ K.T / np.sqrt(d_k)

    # Step 3: Softmax to get weights
    attention_weights = softmax(scores)

    # Step 4: Weighted sum of values
    output = attention_weights @ V

    return output, attention_weights

# Test it
output, weights = self_attention(X, W_Q, W_K, W_V)
print(f"Input shape:  {X.shape}")
print(f"Output shape: {output.shape}")
print(f"Attention pattern shape: {weights.shape}")
```

```
Input shape:  (3, 8)
Output shape: (3, 4)
Attention pattern shape: (3, 3)
```

Context-Aware Meaning

> **How attention solves ambiguity in language**

The Problem: Ambiguous Words

In isolation, the word "bank" is **ambiguous**. It could mean:

- A financial institution ("I went to the bank...")
- The side of a river ("The bank was flooded...")

Static embeddings provide a "blended" average meaning, which is often insufficient for deep understanding.

The Solution: Context Updates

Attention allows the model to **update the representation** of "bank" based on the surrounding tokens.

Disambiguation: By attending to words like "money," "loan," or "river," the model pulls in specific features that **resolve the ambiguity**, creating a context-aware representation.

Context	"bank" attends to...	Final meaning
"I deposited money at the bank"	money, deposited	Financial institution
"We sat on the river bank"	river, sat	Side of a river

A Concrete Walkthrough

> **Tracking "it" through the attention mechanism**

The Example

```
"The animal didn't cross the street because it was too tired."
```

To understand this sentence, the model must resolve the ambiguity of **"it"**. Does it refer to the animal or the street?

Attention Steps

Step	What Happens	Details

Step	What Happens	Details
1. The Query	"it" sends out a Query	"What noun am I referring to?"
2. The Match	Keys are compared	"animal" has a Key that matches the Query. "Street" also matches, but less strongly.
3. The Update	Values are retrieved	The representation of "it" is **updated** by pulling in semantic features from "animal."
4. The Result	Disambiguation complete	The model now "knows" that "it" refers to the animal, enabling correct logical processing.

> [!TIP] This is exactly how transformers perform **coreference resolution** — one of the hardest tasks in NLP!

The Generation Flow

> **How LLMs generate text token by token**

Token-by-Token Generation

```
Step 1: "The"         → compute Q, K, V → attention → next token "cat"
Step 2: "The cat"     → compute Q, K, V → attention → next token "sat"
Step 3: "The cat sat" → compute Q, K, V → attention → next token "on"
And so on...
```

Each step, we run attention over the **ENTIRE sequence**.

What Does the New Token Need?

When generating token 4 ("on"), what do we compute?

Component	For New Token	For Previous Tokens
Query (Q)	☑ Needs fresh computation	❌ Not needed

Component	For New Token	For Previous Tokens
Key (K)	☑ Compute and cache	Already computed — DON'T CHANGE
Value (V)	☑ Compute and cache	Already computed — DON'T CHANGE

> [!IMPORTANT] **Only the NEW token's Q changes each step!** Previous tokens' K and V remain the same.

💾 KV Cache: The Optimization

> **Avoiding redundant computation during generation**

The Waste (Without Cache)

```
For generating "cat":
  Compute: Q_the, K_the, V_the  ← needed
           Q_cat, K_cat, V_cat  ← needed (new token)

For generating "sat":
  Compute: Q_the, K_the, V_the  ← SAME AS BEFORE! (wasted work)
           Q_cat, K_cat, V_cat  ← SAME AS BEFORE! (wasted work)
           Q_sat, K_sat, V_sat  ← needed (new token)
```

We only need Q for the NEW token. But K and V for OLD tokens? **Already computed. Wasted work.**

The Solution: Cache K and V

Instead of recomputing K and V for all tokens... **STORE them after first computation.**

```
Step 1: "The" → compute K_the, V_the → CACHE them
Step 2: "cat" → compute K_cat, V_cat → CACHE them
                use cached K_the, V_the
Step 3: "sat" → compute K_sat, V_sat → CACHE them
                use cached K_the, K_cat, V_the, V_cat
```

Only compute K, V for the NEW token each step. Q is always fresh (only need it for current token anyway).

⏱ Why is the First Token Slow?

When you send a message to ChatGPT:

FIRST token (slow):

- → Process your **ENTIRE prompt**

- → Compute K, V for **every token** in prompt
- → Fill the KV cache
- → Generate first response token

SUBSEQUENT tokens (fast):

- → K, V for prompt **already cached**
- → Only compute for new token
- → Much less work

> [!NOTE] **"Time to first token" and "tokens per second" are different metrics.** Now you know why!

KV Cache Grows with Context

Context Length	Cache Size	Performance
100 tokens	Small cache	Fast
10,000 tokens	Large cache	Slower
100,000 tokens	HUGE cache	Memory problems

- Each layer stores K and V for all tokens
- GPT-4 has **~120 layers**
- Context of 128K tokens = **massive memory**

> [!CAUTION] **This is why long context is hard.** The KV cache grows linearly with context length × number of layers.

Multi-Head Attention

> **One head isn't enough — language has many types of relationships**

The Problem: One Perspective

Single attention head:

- One set of Q, K, V weights
- One "perspective" on relationships

But language has MANY types of relationships:

- **Syntactic:** subject → verb
- **Semantic:** pronoun → referent
- **Positional:** nearby words
- **Topical:** related concepts

One head can't capture all of these.

The Solution: Multiple Heads

```
Head 1: Maybe learns syntax (subject-verb)
Head 2: Maybe learns coreference (it → animal)
Head 3: Maybe learns local context (nearby words)
Head 4: Maybe learns topic relationships
...
```

Each head has its OWN Q, K, V weights. Each head sees different patterns.

How It Works

```
# Multi-head attention (simplified)
num_heads = 8
d_model = 512
d_k = d_model // num_heads  # 64 per head

# Each head has its own projection
heads = []
for i in range(num_heads):
    head_output = self_attention(X, W_Q[i], W_K[i], W_V[i])
    heads.append(head_output)

# Concatenate all heads
multi_head_output = concatenate(heads)  # Back to d_model size
output = multi_head_output @ W_O  # Final projection
```

Model	Number of Heads	Head Dimension
GPT-2	12	64
GPT-3	96	128
GPT-4	(estimated) 128+	~96

Why 8 Heads? (And Not More?)

The Dimension Split:

```
Instead of:
  d_model = 768, one attention with 768-dim Q, K, V

We do:
  8 heads, each with 96-dim Q, K, V (768 ÷ 8 = 96)
```

Each head computes:

```
Q_i = X @ W_Q_i    # produces 96-dim queries
K_i = X @ W_K_i    # produces 96-dim keys
V_i = X @ W_V_i    # produces 96-dim values

output_i = Attention(Q_i, K_i, V_i)
```

Final: Concatenate all heads → project back to 768

The Trade-off

More Heads	Fewer Heads
More perspectives on the data	Fewer perspectives
Each head has smaller dimension	Each head has larger dimension
May capture different relationship types	May capture more nuanced single relationships

The math:

```
8 heads × 96 dims = 768 total
16 heads × 48 dims = 768 total
```

> [!WARNING] **Too many heads → each is too small to capture anything meaningful**
> **Too few heads → not enough perspectives on the data**

Typical in practice: 8-32 heads, depending on model size.

> [!TIP] **Different heads specialize in different relationship types.** This is why transformers are so powerful — they can attend to multiple aspects of meaning simultaneously!

🎯 Summary: The Full Pipeline

```
Text: "The cat sat"
         ↓
    [Tokenization]
         ↓
Token IDs: [464, 3797, 3332]
         ↓
    [Embedding Lookup]
         ↓
Embeddings: [[0.1, -0.2, ...], [0.3, 0.1, ...], [-0.1, 0.4, ...]]
         ↓
    [Add Positional Encoding]
```

```
          ↓
Position-aware embeddings (now distinguishes word order)
          ↓
     [Self-Attention]
          ↓
Contextualized embeddings (each word "knows about" other words)
```

Key Concepts Summary

Stage	Input	Output	Purpose
Tokenization	Raw text	Token IDs	Convert text to numbers (the model's "eyes")
Embedding	Token IDs	Dense vectors	Give tokens meaning (semantic representation)
Positional Encoding	Embeddings	Position-aware embeddings	Preserve word order (solve bag-of-words limitation)
Self-Attention	Position-aware embeddings	Contextualized embeddings	Model relationships (words "talk to" each other)

Key Takeaways

> [!IMPORTANT]
>
> 1. **Tokenization** determines what the model "sees" — understanding token boundaries explains many LLM limitations
> 2. **Embeddings** encode meaning — similar words have similar vectors, and relationships are mathematical offsets
> 3. **Positional Encoding** preserves order — without it, "dog bites man" = "man bites dog" (permutation invariance)
> 4. **Self-Attention** models context — each word attends to relevant words and produces a contextualized representation

---

Resources

Type	Link
Class Notebook	class3b.ipynb
Attention Paper	Attention Is All You Need
Tiktoken	OpenAI Tokenizer
Illustrated Transformer	Jay Alammar's Guide
Positional Encoding Viz	Kazemnejad's Blog

Type	Link
GloVe Embeddings	Stanford NLP

Quick Reference

Key Numbers to Remember

Metric	Value	Context
GPT-4 Vocabulary	~100,277 tokens	`cl100k_base` encoding
GPT-2 Embedding Dim	768	Per-token vector size
GPT-4 Context Limit	128k tokens	Not words!
English Token Ratio	~1.3 tokens/word	For prose
Multi-head Typical	8-32 heads	Depending on model size
KV Cache per Layer	seq_len × d_model × 2	Keys + Values

Key Formulas

Positional Encoding

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))

PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

Attention Formula

Attention(Q, K, V) = Softmax(QK^T / √d_k) × V

Final Token Representation

Input = TokenEmbedding + PositionalEncoding

Multi-Head Dimension Split

d_k = d_model / n_heads

Common Gotchas

Issue	Explanation
LLMs can't count letters	Tokenization splits words at non-character boundaries
Spaces attach to following word	`" world"` is one token, not `"world"`
Token ≠ Word	API billing and context limits use tokens

Issue	Explanation
Embeddings ignore order	That's why we need positional encoding
First token is slow	KV cache is empty; must process entire prompt
Long context = huge memory	KV cache grows with every token × every layer

---

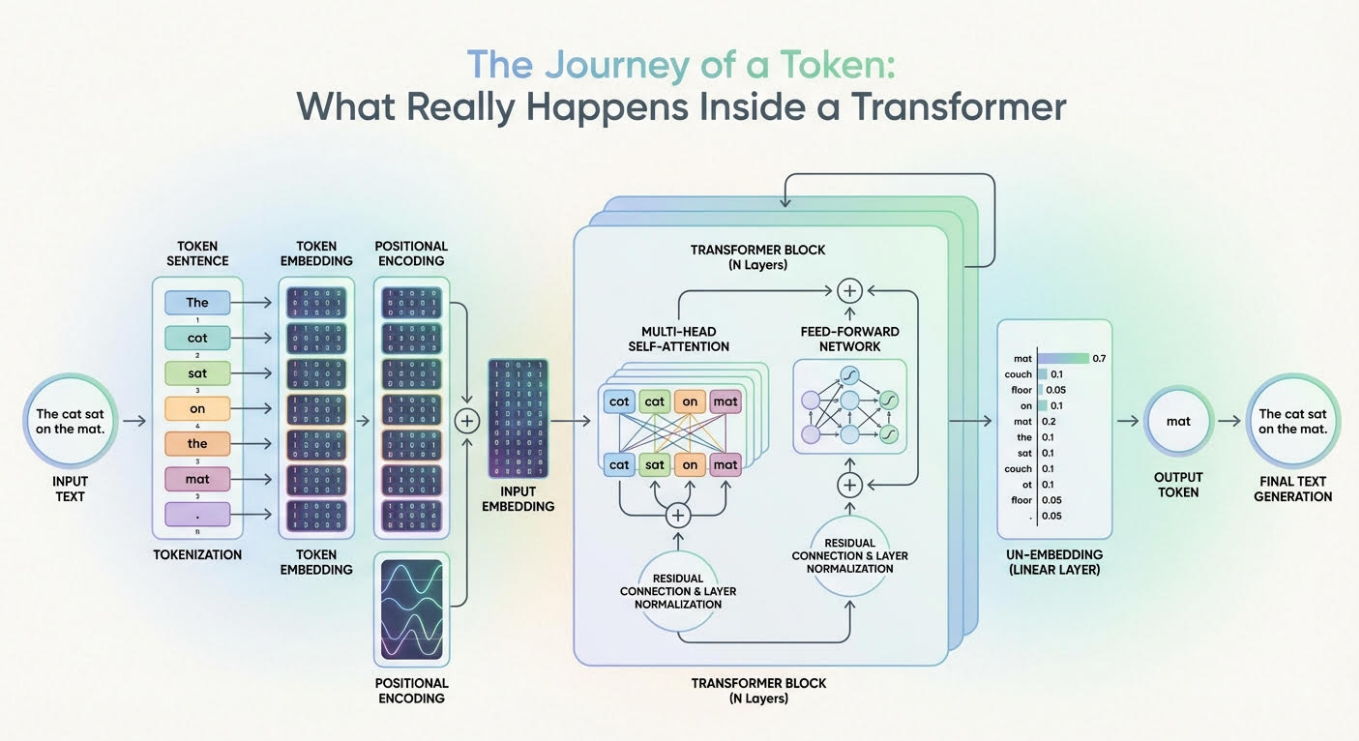
📖 Key Terms Glossary

Term	Definition
Token	Smallest unit of text the model processes
BPE	Byte Pair Encoding — iteratively merges frequent pairs
Embedding	Dense vector capturing semantic meaning
Positional Encoding	Signal added to embeddings indicating position
Self-Attention	Mechanism for tokens to "attend to" each other
Q, K, V	Query, Key, Value — the three projections in attention
Multi-Head Attention	Multiple parallel attention heads with different weights
KV Cache	Stored Keys and Values from previous tokens
Context Window	Maximum tokens processable at once
Permutation Invariance	Order-insensitivity (problem solved by positional encoding)

📆 Class Date: **February 7, 2026**

Made with ❤ by Akshat Shethia during 100xSchool Bootcamp 1.0

The Journey of a Token: From raw text to contextualized understanding! 🚀


The Journey of a Token:
What Really Happens Inside a Transformer
The cat sat on the mat.
INPUT TEXT
TOKEN SENTENCE
The
cat
sat
on
the
mat
.
TOKENIZATION
TOKEN EMBEDDING
TOKEN EMBEDDING
POSITIONAL ENCODING
POSITIONAL ENCODING
INPUT EMBEDDING
TRANSFORMER BLOCK (N Layers)
MULTI-HEAD SELF-ATTENTION
cat cat on mat
cat sat on mat
RESIDUAL CONNECTION & LAYER NORMALIZATION
FEED-FORWARD NETWORK
RESIDUAL CONNECTION & LAYER NORMALIZATION
TRANSFORMER BLOCK (N Layers)
mat 0.7
couch 0.1
floor 0.05
on 0.1
mat 0.2
the 0.1
sat 0.1
couch 0.1
ot 0.1
floor 0.05
. 0.05
UN-EMBEDDING (LINEAR LAYER)
mat
OUTPUT TOKEN
The cat sat on the mat.
FINAL TEXT GENERATION