How Large Language Models Work

From Input Sentences to Token Output

July 5, 2025

Agenda

What are Large Language Models?

- Neural networks trained to understand and generate human language
- Core task: Given input text, predict what comes next
- Enable complex behaviors:
  - Answering questions
  - Writing code
  - Engaging in conversations
- Foundation for RAG (Retrieval-Augmented Generation) systems

The LLM Pipeline Overview

Text → Tokens → Embeddings → Processing → Probabilities → Sampling → Output

Stage	Description
1. Tokenization	Break text into token IDs
2. Embedding	Convert tokens to dense vectors
3. Processing	Transform through neural network layers
4. Prediction	Generate probability distributions
5. Sampling	Select specific tokens
6. Decoding	Convert tokens back to text

Input: "The quick brown fox jumps"
Output: "over the lazy dog"

What is Tokenization?

- Breaks down input text into smaller units called "tokens"
- Tokens can be:
  - Parts of words (subwords)
  - Entire words
  - Punctuation marks
- Modern LLMs use subword tokenization (BPE, SentencePiece)

Example:

Input	"The quick brown fox jumps"
Tokens	["The", " quick", " brown", " fox", " jumps"]
Token IDs	[464, 2068, 2930, 39935, 35308]

Tokenization in Python

```python
def simple_tokenize(text):
    # In reality, this uses sophisticated algorithms like BPE
    tokens = text.split()  # Simplified splitting

    # Each token gets mapped to a unique ID
    vocab = {"The": 464, "quick": 2068, "brown": 2930,
             "fox": 39935, "jumps": 35308}

    token_ids = [vocab.get(token, 0) for token in tokens]
    return tokens, token_ids

tokens, ids = simple_tokenize("The quick brown fox jumps")
print(f"Tokens: {tokens}")
print(f"Token IDs: {ids}")
```

Key Points About Tokenization

- **Vocabulary Size:** Most LLMs have 30K-100K+ tokens
- **Special Tokens:** Markers for beginning/end of text, padding
- **Subword Benefits:**
  - Handle rare words effectively
  - Enable model to work with any text
- **Consistency:** Same text always produces same tokens
- **Language Agnostic:** Works across different languages

Converting IDs to Vectors

- Each token ID $\rightarrow$ dense vector representation (embedding)
- Embeddings capture semantic meaning
- Learned during training process
- Typical dimensions: 768, 1024, 4096, etc.

Example:

Token ID	**Embedding (simplified)**
464 ("The")	[0.1, -0.3, 0.7, ..., 0.2]
2068 ("quick")	[-0.2, 0.8, -0.1, ..., 0.5]

Embedding Implementation

```python
import numpy as np

class TokenEmbedding:
    def __init__(self, vocab_size, embedding_dim):
        # Initialize embeddings (trained in practice)
        self.embeddings = np.random.normal(
            0, 0.1, (vocab_size, embedding_dim))

    def embed(self, token_ids):
        # Look up embeddings for each token
        return self.embeddings[token_ids]

# Example: 50K vocabulary, 768-dimensional embeddings
embedder = TokenEmbedding(vocab_size=50000, embedding_dim=768)
token_ids = [464, 2068, 2930, 39935, 35308]
embeddings = embedder.embed(token_ids)

print(f"Shape: {embeddings.shape}")  # (5, 768)
```

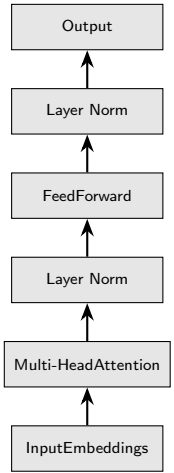
Positional Encoding

- Neural networks don't inherently understand sequence order
- **Solution:** Add positional information to embeddings
- Methods:
  - Sinusoidal encoding
  - Learned positional embeddings
  - Relative position encoding
- Enables model to understand word order and relationships

$$\textbf{Final Input} = \textbf{Token Embedding} + \textbf{Positional Encoding}$$

The Transformer Architecture

- **Multi-Head Attention:** Focus on different parts of input
- **Feed-Forward Networks:** Process information at each position
- **Layer Normalization:** Stabilize training
- **Residual Connections:** Help gradient flow
- **Multiple Layers:** Stack of identical layers (12-96+)



Attention Mechanism

- **Key Innovation:** Allows model to focus on relevant parts
- **Three Components:**
  - Query (Q): "What am I looking for?"
  - Key (K): "What do I contain?"
  - Value (V): "What information do I provide?"
- **Process:**
  1. Calculate attention scores: score $= Q \cdot K^T$
  2. Apply softmax: weights $=$ softmax(scores)
  3. Weighted sum: output $=$ weights $\cdot V$

Simplified Attention Implementation

```python
def simplified_attention(query, key, value):
    # Calculate attention scores
    scores = np.dot(query, key.T)

    # Apply softmax to get probabilities
    attention_weights = softmax(scores)

    # Weighted sum of values
    output = np.dot(attention_weights, value)
    return output, attention_weights

def softmax(x):
    exp_x = np.exp(x - np.max(x))
    return exp_x / np.sum(exp_x)
```

Multi-Layer Processing

- Information flows through multiple transformer layers
- Each layer refines the representation
- Deeper layers capture more complex patterns
- Residual connections preserve information flow

Layer 1: Basic patterns (words, syntax)
Layer 6: Grammatical relationships
Layer 12: Complex semantic understanding
Layer 24+: Abstract reasoning, world knowledge

From Hidden States to Probabilities

- After processing: model produces **hidden states** for each position
- For generation: focus on the **last position**
- Project hidden state to vocabulary size
- Apply softmax to get probability distribution

Next Token	Probability
"over"	0.35
"quickly"	0.20
"."	0.15
"across"	0.12
"through"	0.08
"..."	0.10

Prediction Implementation

```python
def get_next_token_probabilities(hidden_state, output_projection):
    # Project hidden state to vocabulary size
    logits = np.dot(hidden_state, output_projection)

    # Convert to probabilities
    probabilities = softmax(logits)
    return probabilities

# Example
hidden_state = np.random.normal(0, 1, 768)
output_proj = np.random.normal(0, 0.1, (768, 50000))

probs = get_next_token_probabilities(hidden_state, output_proj)
print(f"Probability shape: {probs.shape}")  # (50000,)
print(f"Sum of probabilities: {np.sum(probs)}")  # ~1.0
```

Choosing the Next Token

- Model doesn't always pick highest-probability token
- Different **sampling strategies** create different behaviors:

Strategy	**Description**
Greedy	Always pick most likely token
Random	Sample proportionally to probabilities
Temperature	Adjust randomness with temperature parameter
Top-k	Only consider k most likely tokens
Top-p (Nucleus)	Consider tokens until cumulative probability reaches p

Sampling Strategies Implementation

```python
def sample_next_token(probabilities, method="greedy",
                      temperature=1.0, top_k=None):
    if method == "greedy":
        return np.argmax(probabilities)

    elif method == "random":
        return np.random.choice(len(probabilities),
                                p=probabilities)

    elif method == "temperature":
        adjusted_probs = softmax(np.log(probabilities) / temperature)
        return np.random.choice(len(adjusted_probs),
                                p=adjusted_probs)

    elif method == "top_k" and top_k:
        top_indices = np.argsort(probabilities)[-top_k:]
        # ... implementation continues
```

Temperature Effects

Temperature	Behavior	Use Case
0.1 - 0.5	Focused, deterministic	Factual answers, code
0.7 - 1.0	Balanced creativity	General conversation
1.0+	Random, creative	Creative writing, brainstorming
0	Completely deterministic	Reproducible outputs

Example with "The weather is":

- **Low temp (0.2):** "The weather is sunny and clear today."
- **High temp (1.5):** "The weather is mysteriously whispering secrets."

The Autoregressive Process

- LLMs are **autoregressive**: generate one token at a time
- Use their own output as input for next prediction
- Continue until stopping condition is met

Example Generation:

Step	Sequence
Input	"The weather today is"
Step 1	"The weather today is sunny"
Step 2	"The weather today is sunny and"
Step 3	"The weather today is sunny and warm"
Step 4	"The weather today is sunny and warm."

Text Generation Loop

```python
def generate_text(model, tokenizer, prompt, max_length=50):
    # Tokenize initial prompt
    tokens = tokenizer.encode(prompt)
    for _ in range(max_length):
        # Get model predictions
        probabilities = model.predict(tokens)

        # Sample next token
        next_token = sample_next_token(
            probabilities, method="temperature", temperature=0.7)

        # Add to sequence
        tokens.append(next_token)

        # Check for end-of-sequence token
        if next_token == tokenizer.eos_token_id:
            break
    # Decode back to text
    return tokenizer.decode(tokens)
```

Why This Matters for RAG

- **Context Window Limitations:** LLMs have finite context (4K-128K tokens)
- **Knowledge Cutoffs:** Training data has time limits
- **Domain-Specific Knowledge:** May lack specialized information

RAG Solutions:

- Retrieve relevant information before generation
- Inject retrieved context into input
- Leverage LLM's reasoning with external knowledge

RAG Integration Points

Traditional LLM:

User Query $\rightarrow$ Tokenization $\rightarrow$ Generation $\rightarrow$ Response

RAG-Enhanced LLM:

User Query

$\downarrow$

Document Retrieval

$\downarrow$

Context Injection

$\downarrow$

Augmented Query + Context

$\downarrow$

Tokenization $\rightarrow$ Generation $\rightarrow$ Response

Analyzing RAG Performance

Understanding LLM internals helps with:

- **Attention Analysis:** Visualize which retrieved documents the model focuses on
- **Token Confidence:** Examine probability distributions to detect uncertainty
- **Context Utilization:** Determine if retrieved information is being used effectively
- **Hallucination Detection:** Identify when model generates beyond provided context

Testing Strategies:

- Ablation studies (remove context, measure performance)
- Attention pattern analysis
- Confidence threshold tuning

Complete LLM Pipeline Summary

1. **Tokenization:** Text $\rightarrow$ Token IDs
2. **Embedding:** Token IDs $\rightarrow$ Dense vectors ($+$ positional encoding)
3. **Processing:** Vectors $\rightarrow$ Contextualized representations (Transformer layers)
4. **Prediction:** Representations $\rightarrow$ Probability distributions over vocabulary
5. **Sampling:** Distributions $\rightarrow$ Specific tokens (various strategies)
6. **Decoding:** Tokens $\rightarrow$ Generated text

This process repeats iteratively to generate complete responses

Key Takeaways for RAG Development

- **Context is King:** Everything in the context window influences generation
- **Position Matters:** Recent tokens have more influence than distant ones
- **Attention Patterns:** Show what the model considers important
- **Probability Distributions:** Reveal model confidence and uncertainty
- **Sampling Strategy:** Affects creativity vs. accuracy trade-offs

Next Steps: Learn how RAG systems inject retrieved knowledge into this pipeline to enhance LLM capabilities with external information.