Tokenization and Code Representation

CS450

Understanding how language models process code

What is Tokenization?

Tokenization: Breaking text (including code) into discrete units called **tokens**

- Tokens are the atomic elements models operate on
- Models read, think, and generate in tokens
- Not characters, not words—tokens!

```
"hello world" → [hello] [world]
"getUserById" → [get] [User] [By] [Id]
```

Why Does Tokenization Matter for Code?

Code has unique properties:

- **Precise syntax**: `=` vs `==` matters
- **Meaningful whitespace**: Python indentation
- **Special operators**: `->`, `::`, `>>>`, `**`
- **Case sensitivity**: `userName` vs `UserName`
- **Domain identifiers**: API names, libraries

Consequences of Poor Tokenization

Inefficiency:

- More tokens = less context fits in window
- Higher API costs
- Slower inference

Quality issues:

- Loss of structural information
- Difficulty learning patterns
- Poor generation quality

Byte Pair Encoding (BPE)

The standard approach for modern LLMs

How it works:

1. Start with character-level vocabulary
2. Find most frequent adjacent pair
3. Merge into new token
4. Repeat until vocabulary reaches target size

BPE in Action

Common patterns = fewer tokens:

```
"function"      → 1 token
"funcxzqtion"   → 4 tokens

"def factorial(n):"     → 4 tokens
"def qzxfactorial(n):"  → 8 tokens
```

Key insight: Frequent patterns are more efficient

Code Tokenization Examples

```
def calculate_sum(a, b):
```

→ `def|calculate_sum|(|a|,|b|)|:`

```
function calculateSum(a, b) {
```

→ `function|calculateSum|(|a|,|b|)|{`

Special Tokens for Code Structure

Models use special tokens for boundaries:

- `<|endoftext|>` — End of document
- `<|fim_prefix|>`, `<|fim_suffix|>`, `<|fim_middle|>` — Fill-in-the-middle
- `<|python|>`, `<|javascript|>` — Language markers
- `\n`, `\t` — Whitespace (often separate tokens)

Context Matters

Same incomplete code, different contexts:

```
# Top-level
result = calculate → "result = calculate()"

# Inside function
def process(x):
    result = calculate → "result = calculate(x)"

# In class method
class Processor:
    def run(self):
        result = calculate → "result = calculate(self.data)"
```

Indentation and Whitespace

Python's significant whitespace poses unique challenges

Models must learn that **indentation carries semantic meaning**

```
# Correct
def greet(name):
    print(f"Hello, {name}")
    return name

# Incorrect (indentation error)
def greet(name):
print(f"Hello, {name}")
    return name
```

Token Efficiency Comparison

```
# Verbose: 48 tokens
def calculate_sum_of_squares(input_numbers):
    total_sum = 0
    for individual_number in input_numbers:
        squared_value = individual_number * individual_number
        total_sum = total_sum + squared_value
    return total_sum

# Concise: 17 tokens
def sum_squares(nums):
    return sum(n * n for n in nums)
```

3.5x more tokens for same functionality!

Trade-offs: Verbosity vs Efficiency

Verbose code:

- More readable
- Better documentation
- Consumes more tokens

Concise code:

- Token-efficient
- Fits more in context window
- May be less clear

Balance based on your use case

Cross-Language Tokenization

Code models handle multiple languages:

```
# Python
def add(a, b): return a + b

# JavaScript
function add(a, b) { return a + b; }

# Rust
fn add(a: i32, b: i32) -> i32 { a + b }
```

Tokenizer must efficiently handle diverse syntax

Generation Issue #1: Variable Names

Models prefer common patterns:

```
# Prompt: "variable for user authentication tokens"
auth_token = "..."  # Common abbreviation

# Uncommon/long names fragment into many tokens
# Models avoid or truncate them
```

Why? Tokenization efficiency influences generation

Generation Issue #2: Operator Spacing

```
# Models learn spacing from training data
x = a + b * c     # Standard
x=a+b*c           # Compressed
x = a + (b * c)   # Explicit
```

Inconsistent tokenization → inconsistent spacing

Solution: Use examples in prompts for consistency

Generation Issue #3: Comments

```
# With comments request
def factorial(n):
    """Calculate factorial of n."""
    if n == 0:
        return 1
    return n * factorial(n-1)

# Without comments request
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

Comments are tokenized like code

Practical Implications

Understanding tokenization helps you:

✓ Write more effective prompts
✓ Debug unexpected model behavior
✓ Optimize context window usage
✓ Predict when models will struggle
✓ Design better conventions for AI-assisted development

Key Takeaways

1. **Tokenization is fundamental** — It's how models "see" code
2. **Common = efficient** — Frequent patterns use fewer tokens
3. **Structure matters** — Special tokens encode code boundaries
4. **Token count impacts cost** — Efficiency = lower costs, more context
5. **Understand to optimize** — Better tokenization knowledge = better results

Glossary: Essential Terms

Byte Pair Encoding (BPE): Iterative algorithm merging frequent character pairs

Context Window: Fixed token limit for input + output

Special Tokens: Reserved tokens for metadata/structure (`<|endoftext|>`, `\n`)

Tokens: Atomic units models process (not characters or words)