BPE in GPT-1: Complete Sentence Example

Step-by-step tokenization of a real sentence

🎯 Our Example Sentence

Let's tokenize: **"The running dogs are unhappy"**

I'll show you both:

1. **How BPE was trained** (the learning process)
2. **How GPT-1 applies trained BPE** (the tokenization process)

🏗️ Part 1: How BPE Was Trained (Simplified)

Training Data Sample:

```
"low lower lowest run running runner happy unhappy the dog dogs"
```

Step 1: Initialize with Characters

```
Initial text: "l o w </w> l o w e r </w> l o w e s t </w> r u n </w> r u n n i n g </w> r u n n e r </w> h a p p y </w> u n h a p p y </w> t h e </w> d o g </w> d o g s </w>"

Initial vocabulary: {l, o, w, e, r, s, t, u, n, i, g, h, a, p, y, d, </w>}
```

Iteration 1: Most Frequent Pair

Count pairs:

```
(r,u): 3 times (run, running, runner)
(u,n): 3 times (run, running, runner)
(n,n): 2 times (running, runner)
(p,p): 2 times (happy, unhappy)
(o,g): 2 times (dog, dogs)
...
```

Merge most frequent: `(r,u)` → `ru`

```
BEFORE: "r u n </w> r u n n i n g </w> r u n n e r </w>"
AFTER:  "ru n </w> ru n n i n g </w> ru n n e r </w>"
```

Updated vocabulary: `{l, o, w, e, r, s, t, u, n, i, g, h, a, p, y, d, </w>, ru}`

Iteration 2: Next Most Frequent

Count pairs in current state:

```
(ru,n): 3 times
(n,n): 2 times
(p,p): 2 times
...
```

Merge: `(ru,n)` → `run`

```
BEFORE: "ru n </w> ru n n i n g </w> ru n n e r </w>"
AFTER:  "run </w> run n i n g </w> run n e r </w>"
```

Updated vocabulary: `{..., ru, run}`

Continue for thousands of iterations...

After 40,000 merges, GPT-1's vocabulary includes tokens like:

```
{the, run, ning, ing, happy, un, dog, s, low, er, est, ...}
```

🎯 Part 2: How GPT-1 Tokenizes Our Sentence

Input Sentence: "The running dogs are unhappy"

Step 1: Preprocessing

```
Original: "The running dogs are unhappy"
Lowercase: "the running dogs are unhappy"
Add spaces: " the running dogs are unhappy"
Add end markers: " the</w> running</w> dogs</w> are</w> unhappy</w>"
```

Step 2: Start with Character-Level Split

```
Character tokens: [" ", "t", "h", "e", "</w>", " ", "r", "u", "n", "n", "i", "n", "g", "</w>", " ", "d", "o", "g", "s", "</w>", " ", "a", "r", "e", "</w>", " ", "u", "n", "h", "a", "p", "p", "y", "</w>"]
```

Step 3: Apply BPE Merges in Learned Order

Merge 1: `(t,h)` → `th` (from training)

```
BEFORE: [" ", "t", "h", "e", "</w>", ...]
AFTER:  [" ", "th", "e", "</w>", ...]
```

Merge 2: (th,e) → the (very common word)

```
BEFORE: [" ", "th", "e", "</w>", ...]
AFTER:  [" ", "the", "</w>", ...]
```

Merge 3: (the,</w>) → the</w> (complete word token)

```
BEFORE: [" ", "the", "</w>", " ", "r", "u", "n", "n", "i", "n", "g", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "r", "u", "n", "n", "i", "n", "g", "</w>", ...]
```

Merge 4: (r,u) → ru

```
BEFORE: [" ", "the</w>", " ", "r", "u", "n", "n", "i", "n", "g", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "ru", "n", "n", "i", "n", "g", "</w>", ...]
```

Merge 5: (ru,n) → run

```
BEFORE: [" ", "the</w>", " ", "ru", "n", "n", "i", "n", "g", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "run", "n", "i", "n", "g", "</w>", ...]
```

Merge 6: (n,i) → ni

```
BEFORE: [" ", "the</w>", " ", "run", "n", "i", "n", "g", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "run", "n", "i", "n", "g", "</w>", ...]
```

Wait, let me be more realistic. Let's say the trained BPE has these tokens:

Merge 6: (i,n) → in

```
BEFORE: [" ", "the</w>", " ", "run", "n", "i", "n", "g", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "run", "n", "in", "g", "</w>", ...]
```

Merge 7: (in,g) → ing

```
BEFORE: [" ", "the</w>", " ", "run", "n", "in", "g", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "run", "n", "ing", "</w>", ...]
```

Merge 8: (n,ing) → ning

```
BEFORE: [" ", "the</w>", " ", "run", "n", "ing", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "run", "ning", "</w>", ...]
```

Merge 9: (ning,</w>) → ning</w>

```
BEFORE: [" ", "the</w>", " ", "run", "ning", "</w>", " ", "d", "o", "g", "s", "</w>", ...]
AFTER:  [" ", "the</w>", " ", "run", "ning</w>", " ", "d", "o", "g", "s", "</w>", ...]
```

Continue for "dogs": (d,o) → do, (do,g) → dog, (dog,s) → dogs, (dogs,</w>) → dogs</w>

```
Result so far: [" ", "the</w>", " ", "run", "ning</w>", " ", "dogs</w>", " ", "a", "r", "e", "</w>", ...]
```

For "are": (a,r) → ar, (ar,e) → are, (are,</w>) → are</w>

```
Result so far: [" ", "the</w>", " ", "run", "ning</w>", " ", "dogs</w>", " ", "are</w>", " ", "u", "n", "h", "a", "p",
"p", "y", "</w>"]
```

For "unhappy": (u,n) → un, (h,a) → ha, (ha,p) → hap, (hap,p) → happ, (happ,y) → happy, (un,happy) → unhappy, (unhappy,</w>) → unhappy</w>

Final Result After All BPE Merges:

```
Tokens: [" ", "the</w>", " ", "run", "ning</w>", " ", "dogs</w>", " ", "are</w>", " ", "unhappy</w>"]
```

🔧 Step 4: Clean Up and Convert to IDs

Remove Spaces and Convert to Token IDs:

```
Final tokens: ["the</w>", "run", "ning</w>", "dogs</w>", "are</w>", "unhappy</w>"]

Convert to vocabulary IDs:
"the</w>" → ID: 1234
"run" → ID: 5678
"ning</w>" → ID: 9012
"dogs</w>" → ID: 3456
"are</w>" → ID: 7890
"unhappy</w>" → ID: 2468

Final token sequence: [1234, 5678, 9012, 3456, 7890, 2468]
```

🎯 More Realistic GPT-1 Example

Let me show you what **actually** happens with GPT-1's trained BPE:

Sentence: "The running dogs are unhappy"

Actual GPT-1 BPE tokenization (approximate):

```
Step 1: "The running dogs are unhappy"
Step 2: Apply BPE merges in order learned during training
Step 3: Result: ["The", "Ġrunning", "Ġdogs", "Ġare", "Ġun", "happy"]
```

Where `Ġ` represents the space character (GPT uses this notation).

More detailed breakdown:

```
"The" → ["The"] (common word, single token)
" running" → ["Ġrun", "ning"] (space + run + ning suffix)
" dogs" → ["Ġdogs"] (space + common word)
" are" → ["Ġare"] (space + common word)
" unhappy" → ["Ġun", "happy"] (space + prefix + root word)
```

Final tokens: `["The", "Ġrun", "ning", "Ġdogs", "Ġare", "Ġun", "happy"]`

🧠 Key Insights from This Process

1. Efficiency:

- Original: 24 characters
- BPE tokens: 7 tokens
- Much more efficient than character-level (24 tokens)

2. Semantic Preservation:

- "running" = "run" + "ning" (preserves morphology)
- "unhappy" = "un" + "happy" (preserves prefix meaning)
- "dogs" = single token (common word)

3. Flexibility:

- Can handle any word, even if never seen before
- "superunhappy" would become ["super", "un", "happy"]
- No "unknown word" problem!

4. Language Understanding:

- BPE automatically discovers:
  - Common words ("the", "are", "dogs")
  - Prefixes ("un-")
  - Suffixes ("-ing", "-ning")
  - Roots ("run", "happy")

💡 Why This Matters for GPT-1

Perfect Input for Neural Networks:

```
Character-level: [T, h, e,  , r, u, n, n, i, n, g,  , d, o, g, s, ...]  ← 24 tokens, too granular
Word-level: [The, running, dogs, are, unhappy, ????] ← Unknown word problem
BPE: [The, Ġrun, ning, Ġdogs, Ġare, Ġun, happy] ← 7 tokens, perfect balance!
```

Learning Efficiency:

- **Shorter sequences**: Easier for Transformer to process
- **Meaningful chunks**: Each token carries semantic information
- **No unknowns**: Can handle infinite vocabulary
- **Morphology aware**: Understands word structure

This is why BPE was crucial for GPT-1's success - it created the perfect "vocabulary" that balanced efficiency, meaning, and flexibility! 🚀

🎯 Summary

BPE in GPT-1:

1. **Trained once** on massive text corpus (40,000 merges)
2. **Discovers** common character patterns automatically
3. **Applies learned merges** to any new text
4. **Creates efficient tokenization** that preserves meaning
5. **Enables GPT-1** to understand language at the right granularity

The result: **Perfect input representation for the Transformer to learn language patterns!** 🌟