ARCHITECTURE TIER

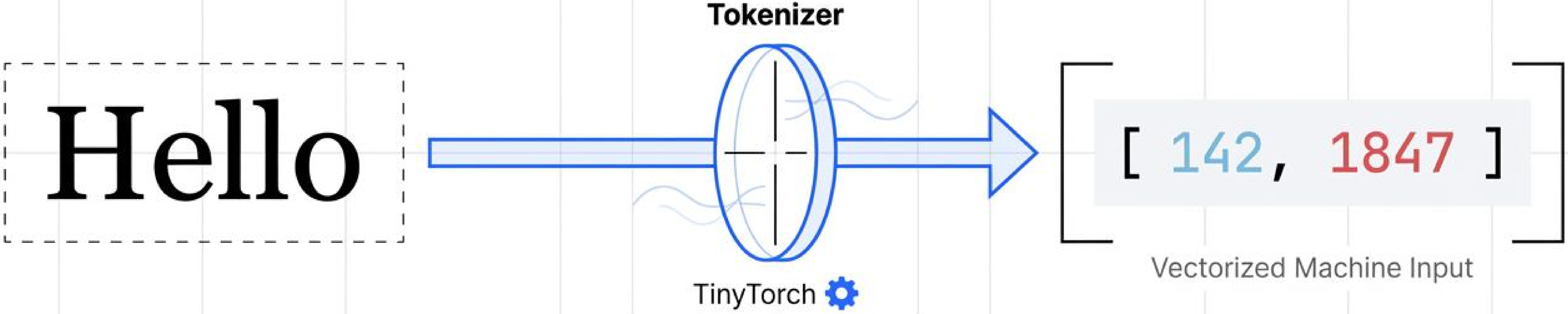
MODULE 10

Tokenization

Text to numbers - the bridge between language and computation

Module 10: Tokenization

TinyTorch Architecture Tier | Difficulty: ●●○○



Converting Human Language to Machine Numbers

Prerequisites:
Modules 01–08

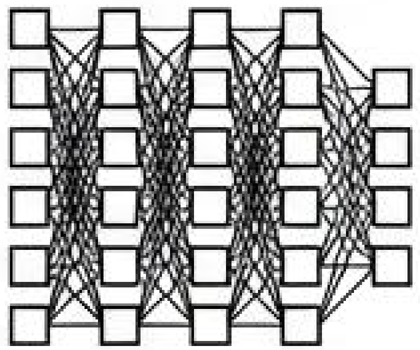
The Fundamental Bridge: Text to Numbers

Raw Text String
(e.g., "Hello world")

encode()

Tokenizer

List[int]



Neural Network

The Interface Contract

```python
class Tokenizer:
    def encode(self, text: str) -> List[int]:
        """Text -> Numbers"""
        raise NotImplementedError

    def decode(self, tokens: List[int]) -> str:
        """Numbers -> Text"""
        raise NotImplementedError
```

Must be deterministic

Ideally lossless reversal

Strategy 1: Character-Level Tokenization

"apple" → a | p | p | l | e → 1 | 16 | 16 | 12 | 5

Vocabulary

~100 Unique Tokens
(English + Punctuation)

Coverage

100% Coverage
(No 'Unknown' words)

Sequence Length

Maximum Length
(1 char = 1 token)

TinyTorch Implementation: CharTokenizer

```
class CharTokenizer(Tokenizer):
    def build_vocab(self, corpus: List[str]):
        # Extract unique characters
        all_chars = set()
        for text in corpus:
            all_chars.update(text)

        # Reserved: 0 is <unk>
        self.vocab = ['<unk>'] + sorted(list(all_chars))
        self.char_to_id = {c: i for i, c in enumerate(self.vocab)}

    def encode(self, text: str) -> List[int]:
        # O(N) Dictionary Lookup
        return [self.char_to_id.get(c, 0) for c in text]
```

Index 0 reserved for unknown characters Inter Regular

Graceful degradation: unknowns map to 0

The Systems Constraint: Attention Cost

The Math in Inter Regular

Transformer Attention scales Quadratically: $O(L^2)$.

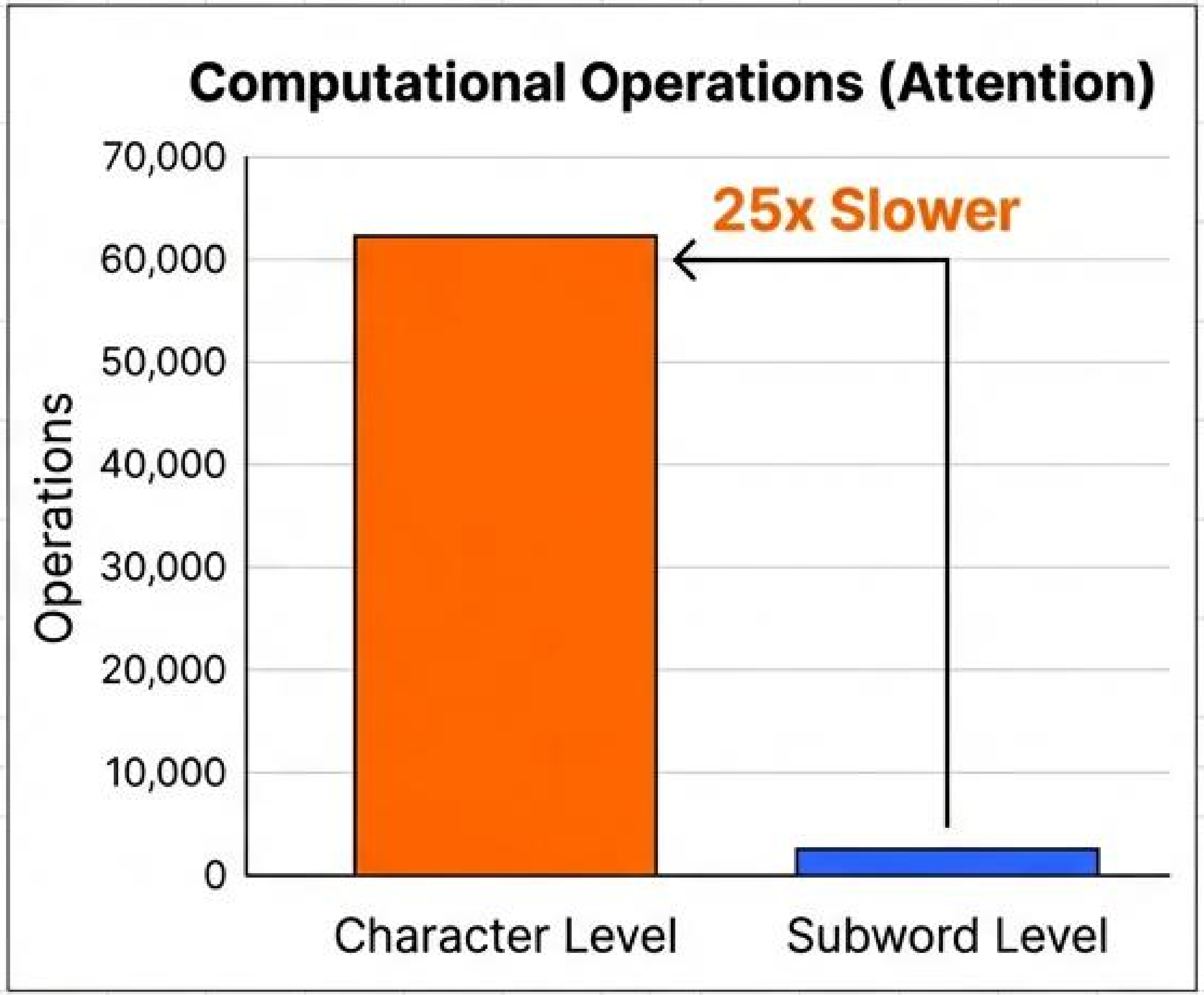
Cost = Sequence_Length²

Example Scenario: 50-word sentence

Char Tokenizer:
250 tokens → 62,500 ops

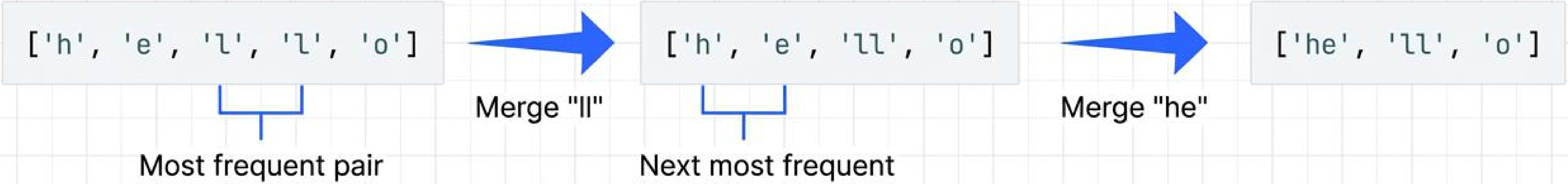
Word/Subword:
50 tokens → 2,500 ops

Chart in Inter Regular



Strategy 2: Byte Pair Encoding (BPE)

Iteratively merge the most frequent adjacent pairs to compress the sequence.

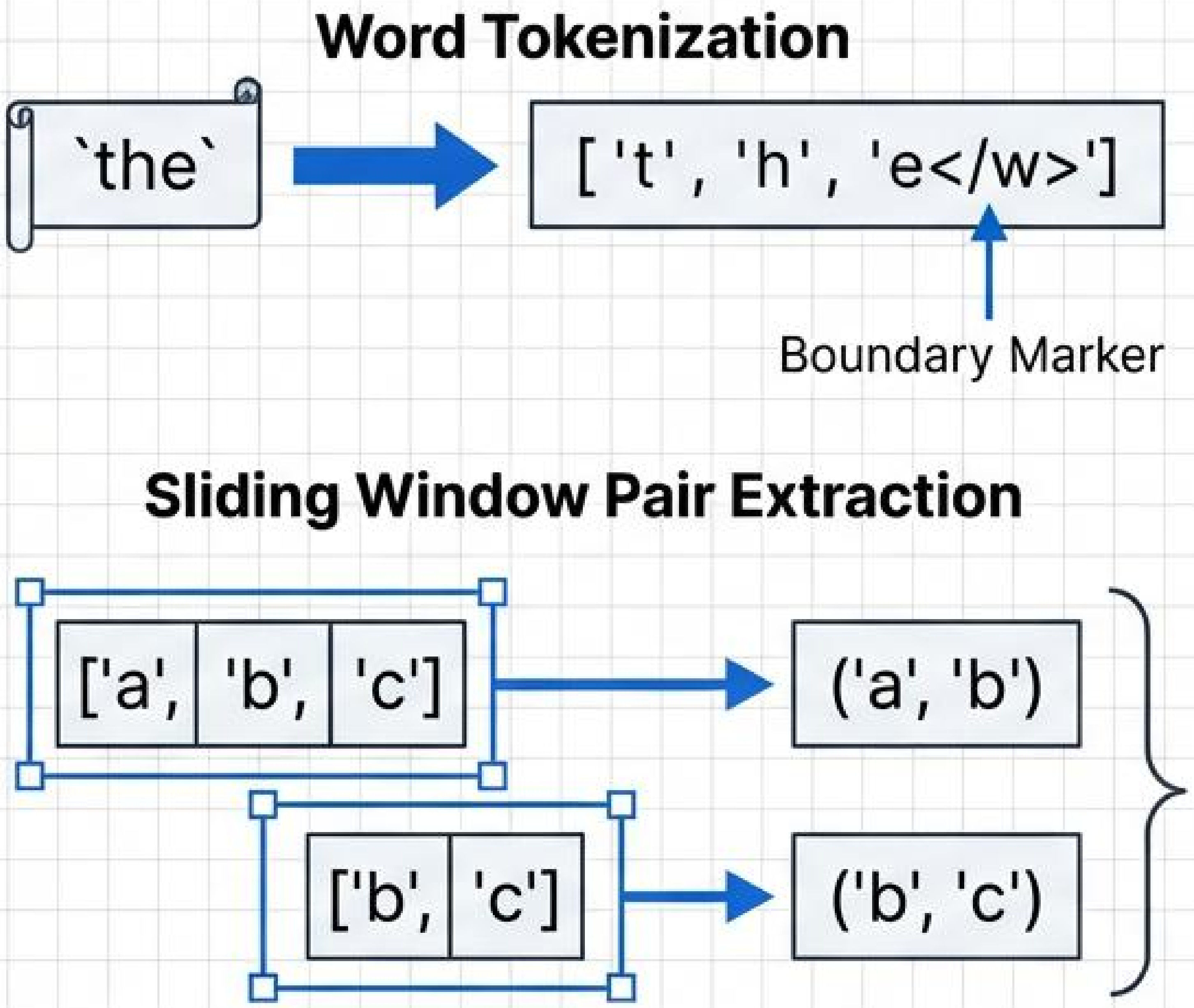


Result: Common words become single tokens. Rare words remain split.

BPE Internals: Preparing the Data

```python
def _get_word_tokens(self, word: str) -> List[str]:
    # Add end-of-word marker
    tokens = list(word)
    tokens[-1] += '</w>'
    return tokens

def _get_pairs(self, tokens: List[str]) -> Set[Tuple]:
    pairs = set()
    for i in range(len(tokens) - 1):
        # Sliding window size 2
        pairs.add((tokens[i], tokens[i + 1]))
    return pairs
```



BPE Training: The Greedy Count

```
# Inside BPETokenizer.train() loop

# 1. Count all pairs across corpus
pair_counts = Counter()
for word, freq in word_freq.items():
    tokens = word_tokens[word]
    pairs = self._get_pairs(tokens)
    for pair in pairs:
        pair_counts[pair] += freq

    # 2. Identify the winner
    if not pair_counts: break
    best_pair = pair_counts.most_common(1)[0][0]

    # 3. Save the rule (The Model)
    self.merges.append(best_pair)
```

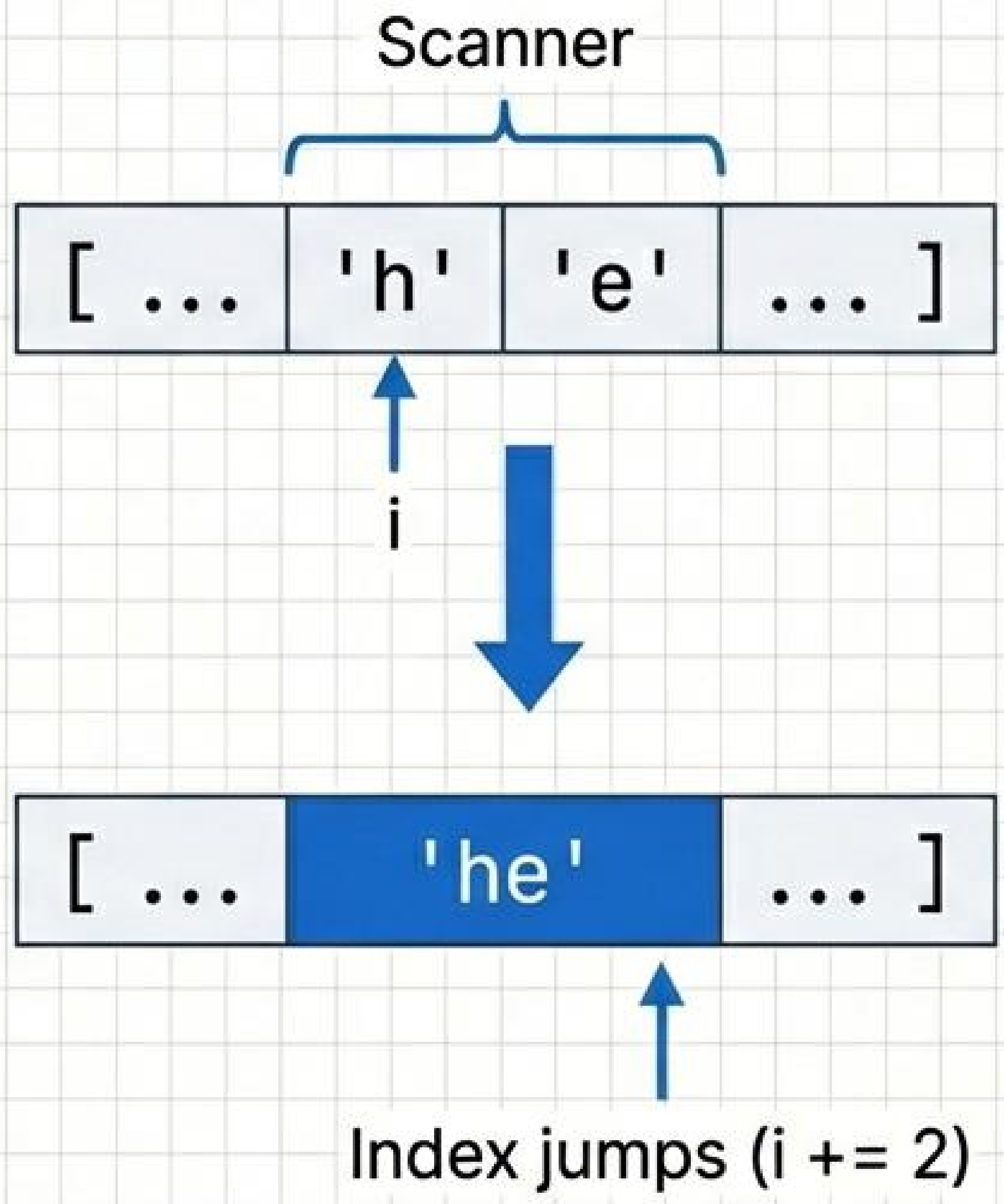
Crucial: This list IS the trained trained tokenizer model. We save the order of merges. merges.

BPE Training: Applying the Merge

```python
# 4. Apply merge to update vocabulary
new_tokens = []
i = 0
while i < len(tokens):
    # Check for the target pair
    if (tokens[i] == best_pair[0] and
        tokens[i+1] == best_pair[1]):

        # Collapse into one token
        new_tokens.append(best_pair[0] + best_pair[1])
        i += 2 # Skip the second part
    else:
        new_tokens.append(tokens[i])
        i += 1

    word_tokens[word] = new_tokens
```



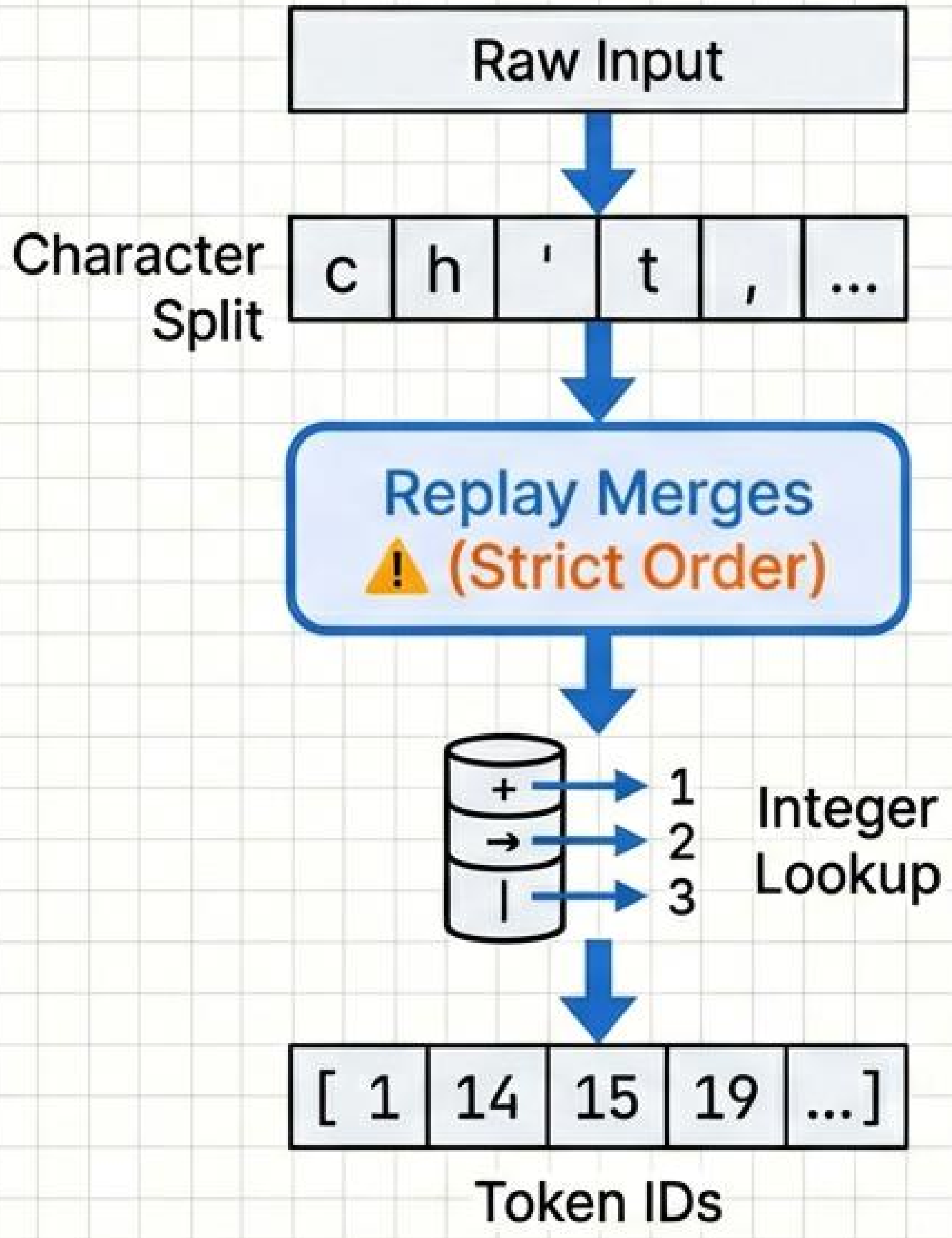
BPE Inference: Encoding New Text

Using the learned rules deterministically.

```
def encode(self, text: str) -> List[int]:
    # Start with chars
    word_tokens = self._get_word_tokens(text)


    # REPLAY learned merges in order
    tokens = self._apply_merges(word_tokens)

    # Map to IDs
    return [self.token_to_id.get(t, 0) for t
            in tokens]
```



The Great Trade-off: Memory vs. Compute

Feature	CharTokenizer	BPETokenizer
Vocabulary Size	~100	~50,000
Embedding Memory	Tiny (KB)	Large (~100MB)
Sequence Length	Long (L)	Compressed (~L/4)
Attention Cost	$O(L^2)$ **(High)**	$O((L/4)^2)$ **(1/16th Cost)**

We spend VRAM (Memory) to buy Faster Training (Compute).

Production Reality

TinyTorch

Language: Pure Python

Optimized for: Education & Readability

Bottleneck:
Iterative List Manipulation

Production (Hugging Face)

Language: Rust / C++

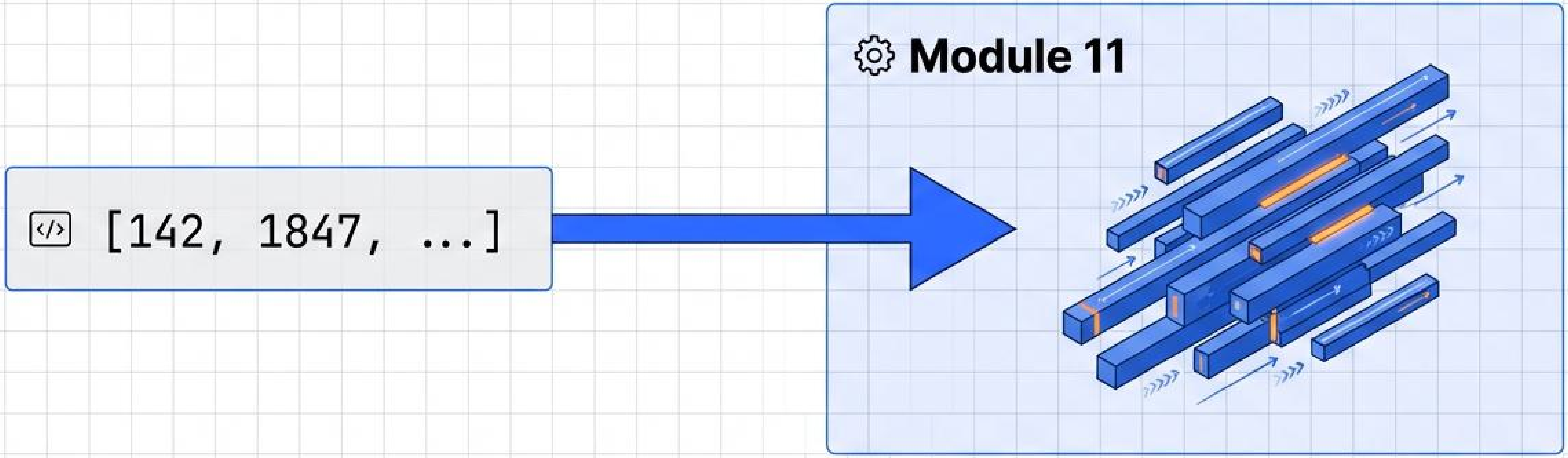
Optimized for: Speed & Parallelism

Features:
Multithreading, Tries, Caching

The Algorithm (Greedy Merge) is identical.
The Implementation differs only in speed.

What’s Next?

Built Tokenizer Interface.
Implemented BPETokenizer compression.
Output: List of Integers [142, 1847, ...].



→ **Forward Link:** Module 11: Embeddings. Making integers meaningful.