The Guts of the Machine: A Deep Dive into LLM Tokenization

Of course. I understand completely. The goal is to create a document that is both a comprehensive, expanded study guide and a faithful representation of the original lecture's narrative, including the speaker's voice, opinions, and the live-coding journey.

Here is the newly revised version, with the personal voice, live coding narrative, and real-time discoveries woven back in.

1 1. The Necessary Evil of Tokenization

Hi everyone. So, in this video, I'd like for us to cover the process of tokenization. You might notice I have a bit of a sad face, and that's because **tokenization is my least favorite part of working with large language models**. Unfortunately, it's necessary to understand in detail because it's a fairly hairy, gnarly field, filled with hidden "footguns" to be aware of. A vast number of the oddities and weird behaviors we see in LLMs can typically be traced directly back to tokenization.

2 2. From Characters to Numbers: A Simple Start

So, what is tokenization? In my previous video, "Let's Build GPT from scratch," we actually already implemented a naive, simple version. We took our training set, the Tiny Shakespeare dataset, which starts as just

a large string of text. To feed it into a neural network, we first built a **vocabulary** of all 65 unique characters present in the text.

Then, we created a simple mapping from each character to an integer—a **token**. For example:

$$\texttt{'H'} \mapsto 20, \quad \texttt{'i'} \mapsto 47, \quad \texttt{' '} \mapsto 1$$

This allowed us to convert any string into a sequence of integers. These integers, in turn, are used as indices to look up a corresponding vector from an **embedding table**. This table is a large matrix where each row represents a token. The row is a vector of numbers (e.g., 384 numbers long) that starts random but is learned via backpropagation. This vector is the rich, numerical representation of the token that is actually fed into the Transformer.

This character-level approach works, but in state-of-the-art models, the schemes are far more complicated. They don't operate on individual characters; they operate on chunks of text, constructed using algorithms like **Byte-Pair Encoding (BPE)**, which we're going to build from scratch today.

3 3. The Source of LLM "Weirdness": Why Tokenization Matters

Before we dive into the code, I want to motivate why this is so important, and frankly, so gross. Tokenization is at the heart of so much weirdness, and I'd advise you not to brush it off. Many issues that look like they belong to the LLM itself are fundamentally tokenization problems.

- **Poor Spelling & String Manipulation**: LLMs often struggle to spell or reverse strings because they see multi-character tokens, not letters.
- **Worse Performance in Non-English Languages**: Tokenizers trained on English-heavy data break other languages into smaller,

less efficient pieces, bloating the token count and eating up the context window.

- **Struggles with Simple Arithmetic**: The way numbers are split into tokens is arbitrary and doesn't align with the digit-by-digit logic humans use.
- **The "Trailing Whitespace" Warning**: That API warning you might have seen is a direct consequence of how spaces are tokenized.
- **The "Solid Gold Magikarp" Glitch**: A famous case where a seemingly innocuous phrase caused an LLM to go completely haywire—a pure tokenization artifact.
- **Code Inefficiency**: GPT-2's poor performance with Python was partly due to its tokenizer turning every single indentation space into a separate token.
- **Format Preference**: The reason developers often suggest **YAML** over **JSON** is token efficiency.

4 4. A Live Exploration: GPT-2 vs. GPT-4 in Action

Let's go to this web app, the Tiktokenizer. It runs tokenization live in the browser. If I type in some text, we can see how the GPT-2 tokenizer splits it.

- A word like "tokenization" becomes two tokens: `token` and `ization`. Notice the leading space is often included, so `␣the` is a single token, distinct from `the`.
- Look at this arithmetic: `127` is one token, but `677` is split into `6` and `77`. It's completely arbitrary.
- Here's another great example: the word "egg."

  - `egg` (lowercase) $\mapsto$ 1 token
  - ` egg` (with a leading space) $\mapsto$ 1 token (but a *different* token ID!)
  - `Egg` (capitalized) $\mapsto$ 2 tokens
  - `EGG` (all caps) $\mapsto$ 3 tokens

  The LLM doesn't see one concept. It sees four different token sequences and has to learn from trillions of examples that they are all related.

- Now I'll paste in some Korean. This phrase is much shorter than the English text above, but it's tokenized into far more pieces. The text is "stretched out" from the model's perspective, making it harder to process.

- Finally, look at this Python code. In GPT-2, every single space for indentation is a separate token (220). This is incredibly wasteful and is a big reason GPT-2 was not great at Python.

Now, let's switch the tool to the GPT-4 tokenizer (`cl100k_base`). The token count for my text drops from 300 to 185! This is a massive efficiency gain. And look at the Python code: four spaces are now grouped into a single, efficient token. This change alone, just in the tokenizer, is a huge part of why GPT-4 is so much better at coding than GPT-2.

5 5. The Bedrock of Modern Tokenizers: UTF-8 and BPE

5.1 5.1. The Encoding Dilemma: Unicode vs. Bytes

To handle all human languages and symbols (like emojis), computers use **Unicode**, which assigns a unique number (**code point**) to ∼150,000 characters.

- **Problem 1**: A 150,000-entry vocabulary is too large and unstable.
- To store these code points, we use an **encoding**. The web standard is **UTF-8**, a variable-length system where common characters take 1 byte and rarer ones take up to 4.
- **Problem 2**: We can't use raw UTF-8 bytes. A byte-level vocabulary only has 256 possible values. This is too small and would create incredibly long sequences, overwhelming the Transformer's finite context length.

So we have a dilemma: a Unicode vocabulary is too big, and a byte vocabulary is too small. The solution that sits in the middle is **Byte-Pair Encoding (BPE)**.

5.2 5.2. The BPE Algorithm: Compressing the Byte Stream

BPE is a simple data compression algorithm. Let's walk through it.

1. **Start with Bytes**: Convert your training text into a sequence of integers from 0–255 using UTF-8.
2. **Find the Most Common Pair**: Count all adjacent pairs of bytes. Find the pair that occurs most frequently.
3. **Merge and Mint**: "Merge" this pair into a single new token. We "mint" a new token ID (starting at 256), add it to our vocabulary, and replace every occurrence of the original pair with this new token.
4. **Repeat**: Go back to step 2. We repeat this for a set number of merges, which defines our final vocabulary size.

6 6. Building a Real-World Tokenizer: A Live Coding Narrative

So, let's now implement all that. I've taken a paragraph from a blog post to use as our toy training text. The first thing we need to do is write a function to iterate through the byte sequence and find the most common pair. I'll call it `get_stats`.

```
# A pythonic way to iterate consecutive elements
for pair in zip(ids, ids[1:]):
    counts[pair] = counts.get(pair, 0) + 1
```

Running this on our text, `get_stats` tells us that the pair (101, 32) is the most frequent, appearing 20 times. What is that? Well, 101 is the byte for `e` and 32 is the byte for a space. So `e ` is the most common pair.

Next, we need a `merge` function that takes this pair and replaces all 20 occurrences with our first new token, ID 256. After implementing that and running it, our original sequence of 616 bytes is now only 596 tokens long. We've compressed it!

Now, we just put this in a loop. For a set number of merges (say, 20 for this example), we find the most common pair and merge it. When I do this on a larger text, after just 20 merges, the original 24,000 bytes are compressed to 19,000 tokens—a **compression ratio of roughly 1.27**.

This whole process—training a `merges` table from a text corpus—is a completely separate, one-time pre-processing stage. The resulting tokenizer is a standalone object from the LLM itself.

7 7. Real-World Complexities: Regex Rules and Special Tokens

The naive BPE we just built isn't quite what's used in production. OpenAI's approach in GPT-2 and GPT-4 adds crucial layers.

7.1 7.1. Beyond Naive BPE: Regex Pre-tokenization

GPT models first apply a **top-down rule** using a complex **regular expression (regex)**. This regex splits the text into chunks *before* BPE runs. Merges are then only allowed to happen *within* these chunks. This is how they prevent illogical merges, like the last letter of a word merging with the first digit of a number.

7.2 7.2. Special Tokens: The Vocabulary's VIPs

Some tokens, like `<|endoftext|>` or chat markers (`<|im_start|>`), are too important to be left to the BPE algorithm. The tokenizer has special-case logic: if it sees the exact string `<|endoftext|>`, it bypasses BPE and outputs token 50256. I have to say, **I don't personally love that OpenAI's original code is in a file called `encoder.py`**, because a tokenizer does both encoding and decoding. It feels a bit awkward.

8 8. An Alternative Philosophy: SentencePiece

Models like Llama and Mistral use a different library, **SentencePiece**. The big difference is that it runs BPE directly on **Unicode code points**, not UTF-8 bytes. For rare characters it hasn't seen, it has a **byte fallback** mechanism. Personally, **I find the Tiktoken way significantly cleaner**. SentencePiece has a lot of historical baggage from machine translation, with confusing concepts like "sentences" and a million settings. In my opinion, the documentation is not super amazing, making it hard to work with.

9 9. The "Footguns" Revisited: Explaining the Weirdness

Now that we understand how this works, let's revisit those weird behaviors:

- **Spelling** `default style`: This phrase is a single token. The model has no concept of the individual letters inside it. It only knows the vector for the whole chunk.
- **Trailing Whitespace**: A trailing space is often a separate, standalone token. The model almost never sees this in its training data (where spaces are usually *prefixes* to the next token) and enters an out-of-distribution state, causing poor performance.
- **The Solid Gold Magikarp**: I'm saving the best for last; this is my favorite one by far.
    1. The **tokenizer's training data** included lots of Reddit text where a user "SolidGoldMagikarp" posted frequently.
    2. BPE merged this common string into a **single, unique token**.
    3. The **LLM's training data**, however, had this Reddit data filtered out.
    4. The `SolidGoldMagikarp` token **never appeared** during the LLM's training. Its embedding vector was never updated; it remained random.
    5. At inference time, typing the string produces the token, which looks up its completely untrained, random vector. Feeding this garbage—effectively **"unallocated memory"**—into the Transformer causes undefined behavior. It breaks the model.

10 10. Final Thoughts and Recommendations

So that concludes my fairly long video on tokenization. I know it's dry, annoying, and irritating. But don't brush it off. There are a lot of footguns, sharp edges, and even security and AI safety issues here.

My final recommendations:

- If you can, reuse the GPT-4 tokenizer via **Tiktoken**. It's efficient and well-designed.
- If you must train your own vocabulary, I would suggest being very careful with **SentencePiece**. Try to copy Meta's settings for Llama exactly, and be aware of its quirks.
- Ideally, what we want is the Tiktoken algorithm but with training code. My **MBP repository** on GitHub is an attempt at this, but it's currently in Python and not yet as efficient as it could be.

Eternal glory goes to anyone who can get rid of tokenization entirely. Until then, it's a part of the stack we must understand.