AI in Colors

From Simple Neural Nets to Large Language Models

DIMITRI REISWICH

8.1 Tokens

Before diving into the inner workings of the neural network itself, we first need to understand how text is represented as input to the model. Language models do not process raw text directly. Instead, they work with smaller building blocks called `tokens`, the atomic units of the model's input.

In an earlier section, we used the bag of words approach, where we split text into words after removing punctuation. In that setting, words served as the atomic units of the language model. These units were then converted into numerical one-hot encoded vectors, which could be fed into a function.

The question now is whether we can use the same idea for a more complex system that processes rich and varied text. At first glance, it might seem reasonable to use the entire English `vocabulary` as the set of atomic units. But this approach comes with several problems. One has to do with you. Another has to do with *Mary Poppins*. If you're from a younger generation, you might need to look her up.

The problem with *you* is that you're human, and like me, you make mistakes, such as typos. When chatting with the model, you might accidentally include a typo, as shown in Figure 8.3. In this example, two mistakes appear in the chat input. The standard English vocabulary does not include the words *draeming* and *abot*, which would cause problems for the model, since the vocabulary is fixed in advance.

Mary Poppins, on the other hand, is known for using the made-up word *supercalifragilisticexpialidocious* in one of her movies. This word is not part of the official English vocabulary, or at least it wasn't before she introduced it. Just like a typo, an unknown word like this would confuse a model that only understands predefined words.

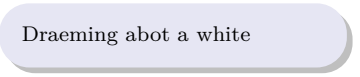

Figure 8.3: Chat input with typos in the words *draeming* and *abot*.

Both of these examples show why using words as atomic units can lead to problems. If words are problematic, the next obvious option would be to use letters as our atomic units. A word is just a sequence of letters, so with letters we could represent any possible word, including completely new ones or words with typos. In that sense, there is nothing fundamentally wrong with using letters.

The problem is efficiency. Every atomic unit needs to be represented inside the neural network, usually as a vector such as a one-hot encoded vector, and each of these representations consumes memory. With letters, the number of units becomes much larger. In our chat example from Figure 8.1, there are only 4 words, but already 19 letters. That means nearly five times as many units to represent.

This matters because modern LLMs often work with long inputs, such as full conversations or documents. If we used letters, the model would need to store and process many more units, increasing both memory usage and computational cost during training and inference. Inference simply means using the trained model to generate new output, such as answering a question or continuing a sentence. So while letters are flexible, they are usually too inefficient for large-scale language models.

So if words and letters don't work, how about choosing something in between, like subwords? This is exactly what a `token` is. It's a chunk of text that can represent different sequences, ranging from individual letters to subwords or even whole words. Tokens strike a balance in terms of computational requirements, sitting right between words and letters. They give us the flexibility of letters when needed, while preserving the efficiency of words whenever possible.

8.1.1 Byte Pair Encoding

Once we decide to use tokens representing subwords as our atomic units, the next question is how to construct those tokens. A simple and intuitive algorithm for doing this is called `byte pair encoding` (`BPE`), which we will explore in this section. Let us walk through a concrete example. We will use a small training text for the algorithm, taken from Wikipedia [62], which wisely takes a moment to appreciate Leonhard Euler's mathematical greatness in the following text:

Euler is regarded as arguably the most prolific contributor in the history of mathematics and science, and the greatest mathematician of the 18th century. Several great mathematicians who worked after Euler's death have recognised his importance in the field. Pierre-Simon Laplace said, "Read Euler, read Euler, he is the master of us all". Carl Friedrich Gauss wrote: "The study of Euler's works will remain the best school for the different fields of mathematics, and nothing else can replace it."

Let us now apply BPE to generate tokens from this text. Note that different variants of the algorithm exist, which differ in implementation details. Here, we follow a commonly used version at the time of writing.

The first step of BPE is to define an initial set of tokens by treating each character in a chosen training text as an individual token. These tokens include letters, punctuation marks such as commas, and any other visible character in the text. These initial tokens for our example text are shown in Figure 8.4. Each small rectangle, with varying widths and gray borders, represents a token and the tokens are arranged in a matrix layout.

Most tokens correspond to individual characters, punctuation marks, numbers, or other symbols. However, some tokens stand out because they are visually longer. These are the result of special handling for characters at the beginning of a word: in these cases, the token includes the preceding space. You can think of this as a kind of pre-merging of two tokens, the white space and the first character of the word, into a single token. This marks the start of a word and allows the algorithm to treat word-initial characters differently from those that appear elsewhere in the text. For example, the first token representing the first character of the word *is* is ␣i, which includes a space and results from merging the two tokens ␣ and i. However, the character *i* later shows up in the word *prolific* as token i, as it is now in the middle of the word. The algorithm treats these cases as different tokens.

The idea of the BPE algorithm is to find the most frequent pair of consecutive tokens and merge, or "mint", them into a new token. You can think of it as joining two letters that often appear together into a new atomic unit. The figure highlights the first such pair in the matrix, Eu, using a blue rectangular box in the top left. The algorithm now moves through the entire matrix, examining each pair of tokens, such as the first one Eu, and counting how often each pair appears in the text.

The counts for the 7 most frequent token pairs in our training text about Euler are

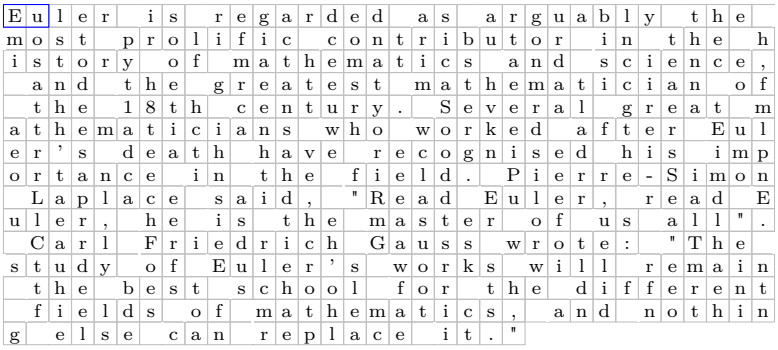

Figure 8.4: Character-level tokenization of the input text, where each character, punctuation, and number is treated as an individual token. The blue rectangular box in the top left highlights the first token pair.

shown in Figure 8.5 as a bar chart. At this stage, the most common pair in the text is h e. An interesting detail in the figure is that the token pairs t h and t h are treated as different. The first pair appears frequently at the beginning of a word, while the second appears in the middle of words such as *mathematician* or *nothing*.

We now select the token pair h e and merge it into a new colored token h e so that it can be easily identified in the matrix. All occurrences of h e in the text are then replaced with this new token, as shown in Figure 8.6.

It's important to note that the original tokens h and e are still part of the text and continue to appear on their own. For example, t appears in the word *different*, and h shows up in *history*. The vocabulary has simply been extended by adding the new token h e.

But what did we actually gain from this? The number of tokens needed to represent the text has gone down. The outer dimensions of the matrix are still the same, so the entire text is still covered. However, the number of rectangles with borders, each representing a token and serving as our atomic unit, has decreased. Originally, there were 13 occurrences of the pair h e, which counted as 26 individual tokens. After introducing the new merged token h e, those 26 tokens are now represented by just 13 single tokens.

There is an important tradeoff here. By adding the merged token h e, the vocabulary has become slightly larger, not smaller. So BPE does not reduce the number of possible tokens the model needs to know. Instead, it reduces the number of token positions needed to write common pieces of text. This is useful because long texts can then be represented as shorter token sequences. In return, the model has to keep track of a larger vocabulary. BPE therefore trades a larger set of possible tokens for shorter sequences of tokens.

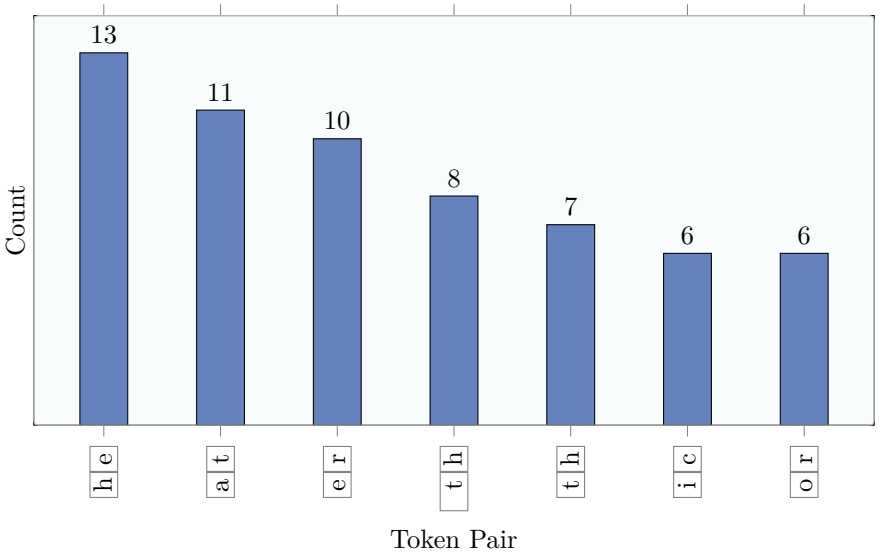

Figure 8.5: Top 7 token pair counts before the first token merge.

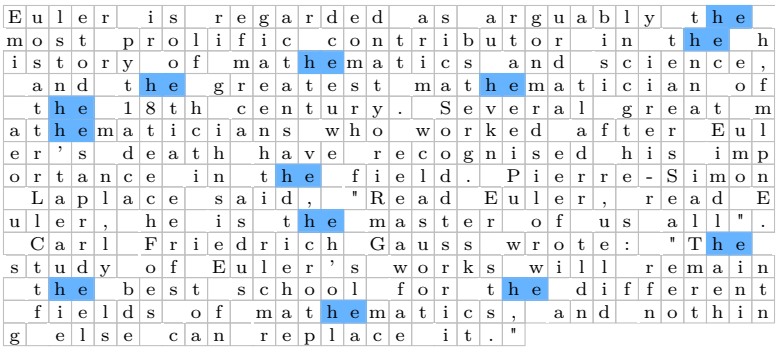

Figure 8.6: Tokens after the first iteration where the token pair h e was merged into a new single token h e.

Let us go through another round. After the first merge, we examine the updated token sequence in Figure 8.6 and count the token pair frequencies again, this time including the newly created token. The result is shown in Figure 8.7. The top token pair is now a t. However, notice how our newly created token h e, in combination with the token t, has already climbed to position 3 in the frequency ranking, just one round after being created. This illustrates that a newly created token is treated in exactly the same way as the original tokens and is immediately available for further merging. The result of merging a t into a newly colored token a t is shown in Figure 8.8. The number of tokens needed to represent our original text has decreased again, while our vocabulary has grown to include one more token.

This process continues, iteration after iteration. The status after 4 iterations is

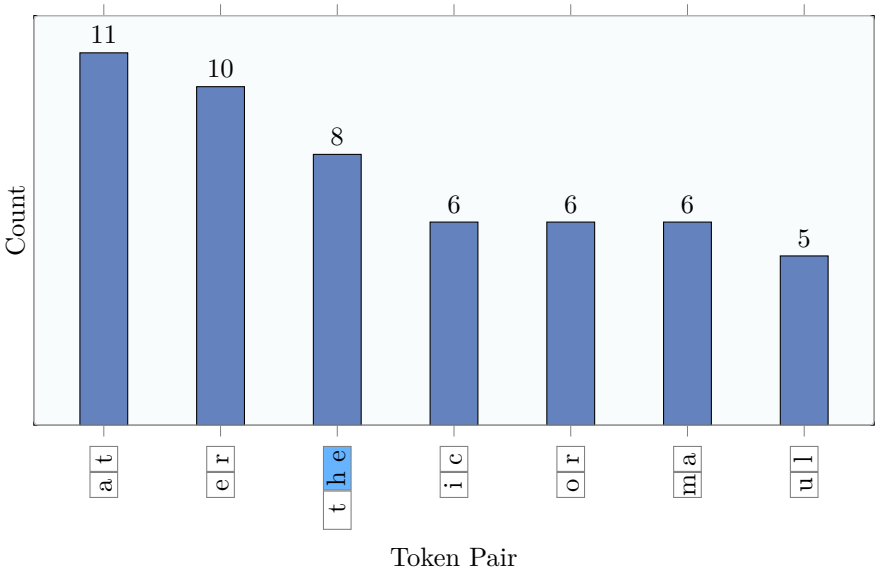

Figure 8.7: Top 7 token pair counts after the first token merge.

Figure 8.8: Tokens after the second iteration where the token pair a t was merged into a single token a t.

shown in Figure 8.9. At this point, we have minted our first token that represents an entire word, namely t h e. After a few more iterations, we can observe that the common subword *uler* emerges as its own token, as shown in Figure 8.10. This highlights how the training dataset directly influences the tokenization. Since the text contains many repetitions of Euler's name, the algorithm learns to treat *uler* as a useful unit. The algorithm keeps merging frequent token pairs until a target vocabulary size, defined by the user, is reached.

Once the tokens have been merged, we can record them in a table and assign each one a numerical ID, as shown in Table 8.1. The original tokens are assigned IDs from 1 to 59, while the newly merged colored tokens start from ID 60. These IDs

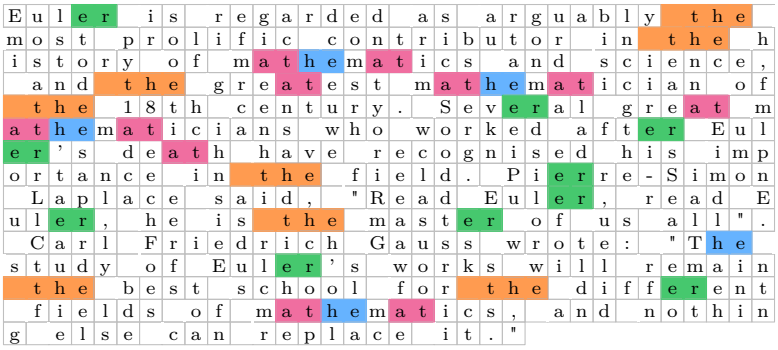

Figure 8.9: Tokens after the fourth iteration.

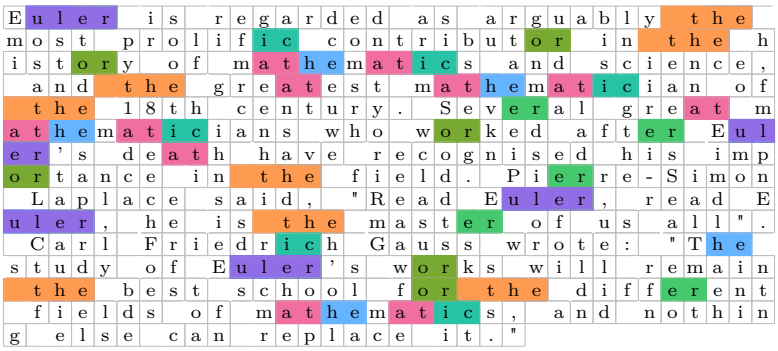

Figure 8.10: Tokens after several iterations.

can now be used to reference the tokens efficiently.

Figure 8.11 illustrates how a tokenized string is converted into a sequence of token IDs. Each token in the string is mapped to a specific ID based on the lookup table in Table 8.1. This step is essential because a language model operates on numerical input rather than raw text. Later, we will see how these token IDs are used inside the model and how they ultimately enable it to process and generate text.

8.1.2 Byte Level Byte Pair Encoding

So far, our tokenization process has focused on standard English vocabulary. Now imagine receiving the message shown in Figure 8.12 from your Spanish friend who lives in the beautiful city of Valencia, Spain. And while your friend didn't mean any harm, she has definitely broken the tokenizer we've discussed so far by simply letting you know how cute the emojis are.

So what is the problem? If we had trained our vocabulary only on standard English text, we wouldn't have encountered special characters like ¡ and é, which are common

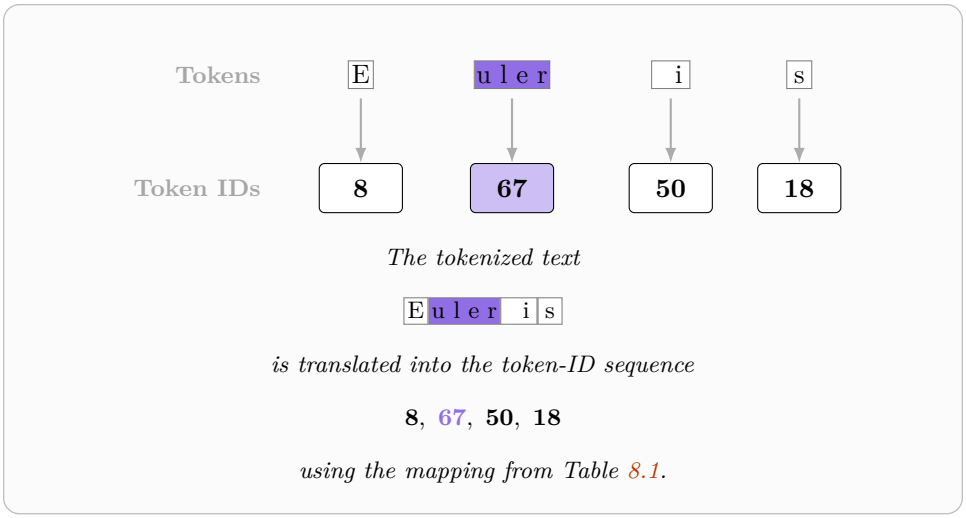

Figure 8.11: Example of converting a tokenized string into its token-ID sequence.

Token	ID	Token	ID
"	1	p	54
’	2	r	55
,	3	s	56
-	4	t	57
.	5	u	58
8	6	w	59
:	7	h e	60
E	8	a t	61
R	9	e r	62
S	10	t h e	63
⋮	⋮	⋮	⋮
		u l e r	67

Table 8.1: Mapping from token to token IDs.

in Spanish. Then there are the panda and koala emojis 🐼 🐨, which aren't letters from any language but symbols. Our tokenizer, trained on a text full of appreciation for Euler, as described so far, wouldn't be able to handle such inputs and would simply break down.

And it doesn't stop there. Think about all the other wonderful languages in the world, like Chinese or Japanese, each with their own writing systems. On top of

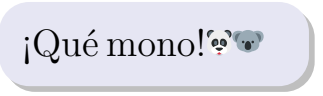

Figure 8.12: A message in Spanish saying "how cute", complete with emojis.

that, we have a huge number of emojis and other special symbols used in digital communication. If we try to include all of these in our vocabulary, as we did in the previous section, it would become extremely large, and the number of tokens would explode. If we don't include them, our model won't be able to handle any input text containing characters it hasn't seen during the tokenizer's training. We need a different approach. That's where an algorithm called `byte-level byte pair encoding` (`BBPE`) comes in. It's similar to the BPE algorithm we discussed, but instead of starting with characters as tokens, it starts with bytes.

To understand BBPE, it helps to first understand how characters and symbols are stored on your computer. A widely used standard for this is called UTF-8. In UTF-8, each character is stored as one or more bytes. A `byte` is a collection of 8 zeros and ones, called `bits`, for example 10010000, and is the basic unit of data storage in most computer systems.

Your computer doesn't directly understand letters or emojis. It reads sequences of bytes and uses standards like UTF-8 to interpret and display them correctly. So when you see an emoji in a message, it's actually the result of your program reading a specific sequence of bytes and rendering it as a symbol on the screen. UTF-8 allows text in different languages, like English, Spanish, or Chinese, and emojis to be represented using variable-length sequences of bytes.

Simple characters, like standard English letters, often take just one byte. For example, the letter Q is stored as the binary sequence 01010001, which is exactly one byte. More complex characters, such as accented letters, symbols, or emojis, may require two, three, or even four bytes. Our cute 🐶, for example, is represented using four bytes: 11110000 10011111 10010000 10111100. You see a panda, the computer sees 32 boring bits.

In standard BPE, we started with characters as the initial tokens. In byte level BPE (BBPE), we go one level deeper and start with bytes instead. This means the text is first converted into its UTF-8 byte representation, and each individual byte becomes its own token. Just like before, each token is assigned a unique numerical ID. Since a byte consists of 8 bits, there are exactly 256 possible byte values, ranging from 00000000 to 11111111. This makes the initial vocabulary simple and fixed: ID 0 corresponds to 00000000, ID 1 to 00000001, and so on, up to ID 255 for 11111111.

The key advantage is that every piece of text, whether it contains English, Spanish, emojis, or any other UTF-8 character, can always be broken down into these bytes. This means BBPE starts with a universal vocabulary that can represent any text before learning larger and more meaningful token combinations through merging.

The characters, their byte representations, and the corresponding token IDs for our input text, including the emojis, are shown in Table 8.2.

Character	Bytes (Tokens)	Token IDs
¡	11000010 10100001	194, 161
Q	01010001	81
u	01110101	117
é	11000011 10101001	195, 169
(space)	00100000	32
m	01101101	109
o	01101111	111
n	01101110	110
o	01101111	111
!	00100001	33
🐼	11110000 10011111 10010000 10111100	240, 159, 144, 188
🐨	11110000 10011111 10010000 10101000	240, 159, 144, 168

Table 8.2: UTF-8 byte and token ID representations of the string ¡Qué mono!🐼🐨.

As a result, our initial text ¡Qué mono!🐼🐨 can now be represented as a list of token IDs as shown in Figure 8.13. The algorithm now continues in the same way as the character-based BPE, starting with single-byte tokens and counting the most frequent token pairs (or token IDs). In our case, the token ID pairs (240, 159) and (159, 144) each appear twice, since both the panda and koala emojis share the same first three bytes in their UTF-8 encoding.

We would then merge the token pair (240, 159) and assign it a new token ID 256, since the first 256 IDs (0 through 255) are already reserved for the individual byte values. After this merge, the 🐼 emoji would be represented by the three tokens (256, 144, 188), and the 🐨 emoji by (256, 144, 168). As a result, our original text is now represented by the following list of token IDs:

194, 161, 81, 117, 195, 169, 32, 109, 111, 110, 111, 33, 256, 144, 188, 256, 144, 168

where the new token has been highlighted in light blue. The algorithm would then continue by merging the most frequent token pairs, which in this case would be the tokens corresponding to (256, 144), just like in standard BPE. This process repeats until a specified vocabulary size is reached.

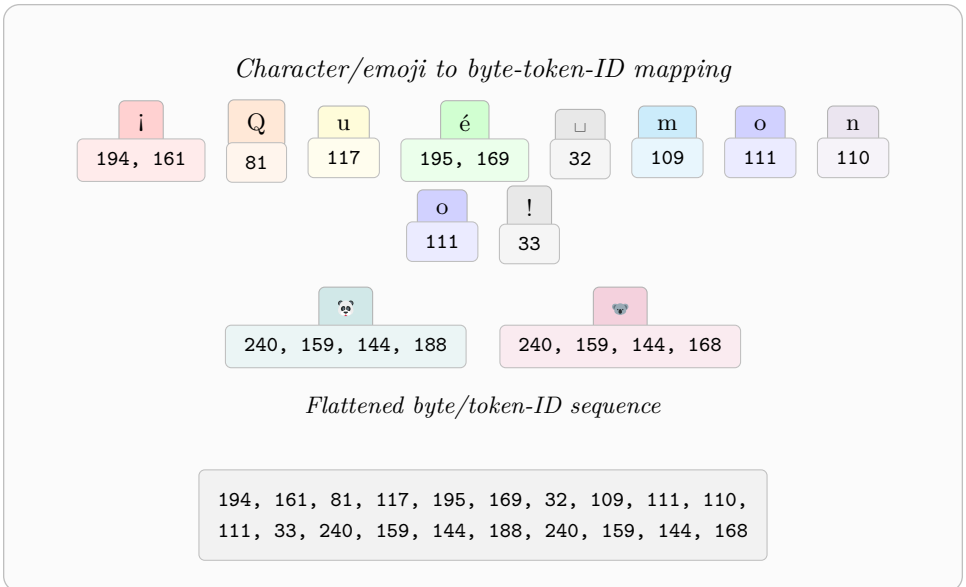

Figure 8.13: Example of representing a text string as a byte-level BPE token-ID sequence.

Other tokenization algorithms can be used too. For example, a commonly used method is Unigram tokenization [29], which we will not cover here.