

Tokenization

Tokenization	1
Tokenization	2
Intro: Tokenization, GPT-2 paper, tokenization-related issues	2
Tokenization by example in a Web UI (tiktokenizerenizer)	3
Strings in Python, Unicode code points	6
Unicode byte encodings, UTF-8, UTF-16, UTF-32	8
Daydreaming: deleting tokenization	9
Byte Pair Encoding (BPE) algorithm walkthrough	9
The implementation	10
Counting consecutive pairs, finding most common pair	11
Merging the most common pair	11
Training the tokenizer: adding the while loop, compression ratio	13
Tokenizer/LLM diagram: it is a completely separate stage	15
Decoding tokens to strings	16
Encoding strings to tokens	18
Regex patterns to force splits across categories	22
Tiktoken library intro, differences between GPT-2/GPT-4 regex	26
Special tokens, tiktoken handling of, GPT-2/GPT-4 differences	29
minbpe exercise time! write your own GPT-4 tokenizer	34
sentencepiece library intro, used to train Llama 2 vocabulary	35
how to set vocabulary set? revisiting gpt.py transformer	40
training new tokens, example of prompt compression	42
revisiting and explaining the quirks of LLM tokenization	44
Final recommendations	50

Intro: Tokenization, GPT-2 paper, tokenization-related issues

Hi everyone, in this video, I'd like us to cover the process of tokenization in large language models. Now, you see here that I have a **sad** face, and that's because tokenization is my least favorite part of working with large language models. But unfortunately, it is necessary to understand in some detail because it is fairly hairy, gnarly, and there are a lot of hidden pitfalls to be aware of. A lot of oddness with large language models typically traces back to tokenization.

What is tokenization? In my previous video, "Let's Build GPT from Scratch," we actually already did tokenization, but we used a very naive, simple version. When you go to the Google Colab for that video, you'll see that we loaded our training set. Our training set was the 'Shakespeare' dataset. In the beginning, the Shakespeare dataset is just a large string in Python, it's just text. So, the question is: How do we plug text into large language models? In this case, we created a vocabulary of 65 possible characters that we saw occur in this string. These were the possible characters, and we observed that there are 65 of them. Then, we created a lookup table for converting every possible character, a string piece into a token—an integer. For example, we tokenized the string 'hii there' and received this sequence of tokens. We took the first 1,000 characters of our dataset and encoded them into tokens. Since this is character-level, we received 1,000 tokens in a sequence (e.g., token 18, 47, etc.). Later, we saw that the way we plug these tokens into the language model is by using an embedding table. Basically, if we have 65 possible tokens, then this embedding table will have 65 rows and roughly speaking, we're taking the integer associated with every single token and that integer using it as a lookup into this table and we're plucking out the corresponding row. This row contains trainable parameters that we're going to train using backpropagation and this is the vector that then feeds into the Transformer, and that's how the Transformer perceives every single token.

Here, we had a very naive tokenization process that was a character-level tokenizer. However, in practice, state-of-the-art language models use much more complicated schemes. Unfortunately, when constructing these token vocabularies, we're not dealing at the character level; instead, we're dealing at the chunk level. The way these character chunks are constructed involves algorithms such as the **byte pairing (2:46)** encoding algorithm, which we'll delve into in detail and cover in this video.

I'd like to briefly show you the paper that introduced byte-level encoding as a mechanism for tokenization in the context of large language models. I would say that's probably the GPT-2 paper. If you scroll down to the section on input representation, you'll find their coverage of tokenization and the desired properties of tokenization. They conclude that they're going to have a tokenizer with a vocabulary of 50,257 possible tokens, and the context size will be 1,024 tokens. In the attention layer of the Transformer neural network, every single token attends to the previous tokens in the sequence, allowing it to see up to 1,024 tokens. Tokens are the fundamental unit—the atom—of large language models. Everything is in units of tokens, and tokenization is the process of translating strings or text into sequences of tokens, and vice versa, when you go into the Llama-2 paper as well, I can show you that when you search 'token,'

you're going to get 63 hits. That's because tokens are, again, pervasive. Here, they mentioned that they trained on two trillion tokens of data and so on.

We're going to build our own tokenizer. Luckily, the byte-pair encoding algorithm is not super complicated, and we can build it from scratch ourselves. We'll see exactly how this works

Before we dive into code I'd like to give you a brief taste of some of the complexities that come from tokenization because I want to make sure we sufficiently motivate why we're doing all this and why it's so gross. Tokenization lies at the heart of a lot of weirdness in large language models. I advise against brushing it off. Many issues that may seem like problems with the new network architecture or the large language model itself actually trace back to tokenization. For instance, if you've noticed any difficulties with large language models handling spelling tasks, that's usually due to tokenization. Simple string processing can be challenging for the large language model to perform natively. Non-English languages can work much worse, and this is largely due to tokenization. Sometimes, LLMs struggle with simple arithmetic, and that can also be traced back to tokenization. Specifically, GPT-2 would have encountered more issues with Python than future versions of it due to tokenization. There are many other issues as well. Perhaps you've seen weird warnings about trailing white spaces—those are tokenization issues.

If you had asked GPT earlier about 'solid gold Magikarp' and what it is, you would see the LLM go totally crazy, veering off into a completely unrelated tangent. Maybe you've been told to use YAML over JSON in structured data—all of that has to do with tokenization. So, basically, tokenization is at the heart of many issues. I will revisit these topics at the end of the video, but for now, let me just skip over them a little bit.

Tokenization by example in a Web UI (tiktokenizerenizer)

Let's go to this web app, <https://tiktokenizer.vercel.app>. I have it loaded here. What I like about this web app is that tokenization is running live in your browser in JavaScript. You can simply type in text—"Hello, world"—and see the entire string tokenized.

"tokenization is " **at** " the " heart of much weirdness of LLMs"

On the left, you'll find the input string. On the right, we're currently using the GPT-2 tokenizer. The string I pasted here is currently tokenizing into **300 tokens**, which are explicitly shown in different colors for each individual token. For example, the word 'tokenization' became two tokens: token 30,642 and 1,634. The token for space is 318. Be mindful that there are spaces and newline characters in the input, but you can hide them for clarity. The token for "space **at**" is 379, and the token for 'space the' is token 262. (space as in the gap between the words)

Notice that the space is part of the token chunk. This breakdown mirrors how our English sentences are segmented. So far, everything seems straightforward.

127 + **677** = **884** (each color representing separate token chunks)

Now, let's explore some arithmetic. We have the token **127** (the token chunk is **127**) followed by 'Plus' and then six (representing another token chunk). That followed by 77. Here's what's happening: 127 feeds in as a single token into the large language model, but the number 677 will actually feed in as two separate tokens. (**6** **77**) The large language model must account for this and process it correctly within its network. Similarly, 804 will be broken up into two tokens. It's all quite arbitrary.

1275 + 6773 = 8041

Here's another example involving four-digit numbers. Their segmentation is also arbitrary—sometimes multiple digits form a single token, while other times individual digits become separate tokens. the main point being that its segmentation is arbitrary. Heres another example

Egg.

I have an **Egg.**

we have the string "Egg." and we see here that it became two tokens. But for some reason when I say "I have an Egg." Egg becomes a single token despite being the exact same string.

egg.

EGG.

Here, lowercase egg turns out to be a single token, and in particular, the color is different so it's a different token meaning that it's case-sensitive. And of course, capital EGG would be different tokens. so for the same concept "egg" depending on if its in the beginning of a sentence, end of the sentence, lowercase or uppercased or mixed, all of this will be different tokens and different IDs and the language model has to learn from raw data from all the internet text that it's going to be trained on that these are actually all the same concept and it sort of has to group them in the parameters of the neural network and understand just based on the data patterns that these are all very similar, albeit not exactly similar.

"**만나서 반갑습니다**" (Manaso Pang, etc.) translates to "Nice to meet you" in Korean.

you'll notice here in korean that non-english languages work slightly worse in Chat-GPT. This is because the training data set is much larger for english. But the same is true, not for the large language model itself, but also for the tokenizer. So when we train the tokenizer we're going to see that there's a training set as well and there's a lot more english than non-english. What ends up happening is that we're going to have a lot more longer tokens for english. if you have a single sentence in english and you tokenize it you might see that it has 10 tokens or so. However, if you translate that sentence into, say, Korean or Japanese (or something else), you'll typically see that the number of tokens used is much larger. That's because the chunks here are a lot more broken up. So, we're using a lot more tokens for the exact same thing. And what this

does is it bloats up the sequence length of all the documents. Consequently, you end up using more tokens. Then, in the attention of the Transformer, when these tokens try to attend each other, you run out of context in the maximum content length of that transformer. Essentially, all the non-English text is stretched out from the perspective of the Transformer. This behavior is related to the training data used for the tokenizer and the tokenization process. As a result, it creates much larger tokens and larger groups in English, while having many smaller boundaries for all the other non-English text. Now, if we translated this into English, it would result in significantly fewer tokens.

```
for i in range(1, 101):  
    if i % 3 == 0 and i % 5 == 0:  
        print("Fizzbuzz")  
    elif i % 3 == 0:  
        print("Fizz")
```

The final example I have here is a little snippet of Python for doing **Fizzbuzz**. What I'd like you to notice is that all these individual spaces are separate tokens. They are token **220** each. And then, the "space if" (`if`) is a single token. So, what's going on here is that when the Transformer consumes or tries to create this text, it needs to handle all these spaces individually. They all feed in one by one into the entire Transformer in the sequence. Unfortunately, this tokenization approach is extremely wasteful. As a result, **GPT-2** is not very good with Python. It's not anything to do with coding or the language model itself. It's just that if we use a lot of indentation using spaces in Python (like we usually do), you end up bloating out all the text. It's separated across way too much of the sequence, and we're running out of the context length in the sequence. That's roughly what's happening. We're being way too wasteful; we're taking up way too much token space.

Now, let's scroll up and consider changing the tokenizer. Note that the GPT-2 tokenizer creates a token count of **300** for this string. However, we can switch to the **CL 100K base** tokenizer, which is the GPT-4 tokenizer. As a result, the token count drops to **185**. So, for the exact same string, we now roughly have half the number of tokens.

This change is beneficial because the same text is now squished into fewer tokens. Consequently, it provides a denser input to the Transformer. Within the Transformer, each individual token has a finite number of tokens before it that it pays attention to. Essentially, we're now able to see roughly twice as much text as context for predicting the next token due to this adjustment. However, it's essential to recognize that merely increasing the number of tokens isn't strictly better indefinitely. As you increase the number of tokens, your embedding

table grows larger. Additionally, at the output, when predicting the next token, the softmax layer also expands. We'll delve into more detail on this topic later, but there's a sweet spot somewhere where you have just the right number of tokens in your vocabulary—where everything is appropriately dense and still efficient.

Now, one thing I would like you to note specifically for the GPT-4 tokenizer is that the handling of white space for Python has improved significantly. Observe the following:

```
for i in range (1, 101):
```

```
    if i % 3 == 0 and i % 5==0:
```

```
        print("Fizzbuzz")
```

```
    elif i % 3 == 0:
```

```
        print("Fizz")
```

you see that here these four spaces are represented as one single token for the three spaces here () and then the token "space if" and here () seven spaces were all grouped into a single token so we're being a lot more efficient in how we represent Python and this was a deliberate choice made by OpenAI when they designed the gpt-4 tokenizer and they group a lot more white space into a single character. What this does is this densifies Python and therefore we can attend to more code before it when we're trying to predict the next token in the sequence. So the improvement in the python coding ability from gpt-2 to gpt-4 is not just a matter of the language model and the architecture and the details of the optimization but a lot of the improvement here is also coming from the design of the tokenizer and how it groups characters into tokens.

Now, let's dive into some code!

Strings in Python, Unicode code points

So, remember what we want to do: we want to take strings and feed them into language models. For that, we need to somehow tokenize strings into integers using a fixed vocabulary. Then, we'll use those integers to look up vectors in a lookup table and feed those vectors into the Transformer as input. Now, the reason this gets a little bit tricky is that we don't just want to support the simple English alphabet. We want to support different kinds of languages. For instance, this is “안녕” (an-nyeong) in Korean, which means “hello.” Additionally, we want to handle various special characters that we might find on the internet—like emojis.

So, how do we feed this text into Transformers? Well, what is this text anyway in Python? If you go to the documentation of a string in Python, you'll find that strings are immutable sequences of Unicode code points. But what are Unicode code points?

Unicode code points are defined by the Unicode Consortium as part of the Unicode standard. Essentially, it's a definition of roughly **150,000 characters** right now. These characters span **161 scripts**. If you scroll down([here](https://en.wikipedia.org/wiki/Unicode) <https://en.wikipedia.org/wiki/Unicode>), you'll see that the standard is very much alive—the latest standard being **15.1** as of September 2023. Essentially, this standard defines various types of characters across different scripts.

To access the Unicode code point for a single character in Python, you can use the `ord` function. For example, if I pass in `ord('H')`, I can see that the Unicode code point for the single character 'H' is **104**.

However, this can get arbitrarily complicated. For example, let's consider our emoji here, and we can see that the code point for this one is **128,000**. Alternatively, we can take 'un,' which corresponds to **50,000**. Now, keep in mind that you can't plug in strings here because this doesn't have a single code point. It only takes a single Unicode code point character and tells you its integer value. In this way, we can look up all the characters of this specific string and their code points. By iterating over each character 'X' in this string, we obtain this encoding:

```
[128,000, 50,000, ...]
```

Now, you might wonder: Why can't we simply use these integers directly without any tokenization? Why not just use the code points as they are? Well, there are a couple of reasons:

1. **Vocabulary Length:**

- If we used raw code points directly, the vocabulary would be quite extensive. For Unicode, this would mean a vocabulary of **150,000** different code points.
- Managing such a large vocabulary could be challenging.

2. **Stability and Change:**

- The Unicode standard is very much alive and keeps evolving.
- Using raw code points might not provide a stable representation, especially if the standard changes over time.

For these reasons, we need a better approach. Let's explore Unicode byte encodings, including UTF-8, UTF-16, and UTF-32. Encodings are the way by which we can take Unicode text and translate it into binary data or byte streams.

Unicode byte encodings: UTF-8, UTF-16, UTF-32

If we visit the Wikipedia page on Unicode encodings, we find that the Unicode Consortium defines three primary encodings: **UTF-8**, **UTF-16**, and **UTF-32**. These encodings allow us to translate Unicode text into binary data or streams. Among these, **UTF-8** is by far the most common. **(124,124,642,346,23)** Let's delve into the details:

1. UTF-8:

- UTF-8 takes each Unicode code point and translates it into a byte stream.
- The length of this byte stream varies between **one to four bytes**.
- It's a variable-length encoding, depending on the Unicode point according to the schema.

For each code point, you'll end up with between **1 to 4 bytes**. Additionally, there are **UTF-16**, and **UTF-32**. Let's explore these encodings:

1. UTF-16:

- UTF-16 uses 2 bytes per character primarily.
- However, it starts to use 3 or more bytes for higher-order characters.
- One disadvantage is that it introduces zero bytes (Z) between characters.

2. UTF-32:

- UTF-32 covers all possible characters in 4 bytes.
- It's fixed-length but bloated compared to the other two.
- Not commonly used due to its inefficiency.

The **UTF-8 Everywhere Manifesto** explains why UTF-8 is significantly preferred and widely used on the internet. One major advantage is its backward compatibility with the simpler ASCII encoding. While UTF-16 and UTF-32 have their merits, UTF-8 remains the go-to choice for many applications.

Remember that choosing the right encoding depends on efficiency, compatibility, and stability. Now, let's encode a string using UTF-8 in Python. The `encode()` method returns a bytes object, which we can inspect further. Also, let's look at **UTF-16**. We get a slightly different byte stream, and here we start to see one of the disadvantages of UTF-16. When we look at the encoded string with UTF-16, Notice how we have zeros (**0, 111, 0, 32, 0, 111, 0, 32, 0, 75, 0, 111, 0, 114, 0, 101, 0, 97, 0**). We're starting to get a sense that this is a bit of a wasteful encoding. Indeed, for simple ASCII characters or English characters, we just have the structure of 'zero something, zero something'. It's not exactly nice. The same holds for **UTF-32**. When we expand this, we can start to see the wastefulness of this encoding for our purposes. There are a lot of zeros followed by something, and this is not desirable.

(0, 0, 0, 111, 0, 0, 0, 114, 0, 0, 0, 52, 0, 0, 0, 523, 0, 0, 0)

Suffice it to say that we would like to stick with UTF-8 for our purposes. However, if we just use UTF-8 naively, these are byte streams, that would imply a vocabulary length of only **256 possible tokens**. But this vocabulary size is very, very small. What this would do if we used it naively is that all of our text would be stretched out over very long sequences of bytes. Our sequences would be very long, and remember that we have a finite context length and attention that we can support in a transformer for computational reasons. So, while we only have as much context length, we now have very, very long sequences. This inefficiency won't allow us to attend to sufficiently long text for the purposes of the next token prediction task.

So, we don't want to use the raw bytes of the UTF-8 encoding. We want to be able to support a larger vocabulary size that we can tune as a hyperparameter. But we still want to stick with the UTF-8 encoding of these strings. So, what do we do? Well, the answer, of course, is we turn to the **Byte Pair Encoding (BPE) algorithm**, which plays a crucial role in compressing byte sequences. But before we delve into that, I want to address the desire to feed raw byte sequences directly.

Daydreaming: deleting tokenization

I just want to briefly speak to the fact that I would love nothing more than to be able to feed raw byte sequences into language models. In fact, there's a paper about how this could potentially be done from last summer. Now, the problem is that you actually have to go in and modify the Transformer architecture. As I mentioned, you'll encounter a problem where attention becomes extremely expensive due to the long sequences.

In this paper, they propose a hierarchical structuring of the Transformer that could allow you to feed in raw bytes directly. At the end, they state that these results establish the viability of tokenization-free autoregressive sequence modeling at scale. Tokenization-free would indeed be amazing—we could just feed byte streams directly into our models. Unfortunately, I don't know if this has truly been proven out yet by sufficiently many groups and at a sufficient scale. However, something like this would be incredible, and I hope someone explores it further. For now, we need to compress the text using the Byte Pair Encoding algorithm.

Byte Pair Encoding (BPE) algorithm walkthrough

So, let's see how this works. As I mentioned, the Byte Pair Encoding algorithm is not all that complicated, and the Wikipedia page is actually quite instructive in terms of the basic idea. Here's what we're doing: we have some kind of input sequence. For example, here we have only four elements in our vocabulary: **a**, **b**, **c**, and **d**. We also have a sequence of them (aaabdaaabc). Instead of bytes, let's consider a vocabulary size of four. However, the sequence is too long, and we'd like to compress it.

So, what do we do? We iteratively find the pair of tokens that occur most frequently. Once we've identified that pair, we replace it with a single new token that we append to our vocabulary. For instance, in this example, the byte pair **aa** occurs most often. So, we mint a new token—let's call it **z**—and replace every occurrence of **aa** with **z**. Now we have two **z**'s here (**zabdzabac**). By doing this, we've taken a sequence of 11 characters with a vocabulary size of four and converted it into a sequence of only nine tokens. But now our vocabulary has five elements because we've created a fifth one: **z**, representing the concatenation of **aa**.

We can repeat this process. Let's say we look at the sequence again and identify the pair of tokens that are most frequent. Suppose that pair is now **ab**. We'll replace **ab** with a new token—let's call it **y**. So, **y** becomes **ab**, and every occurrence of **ab** is now replaced with **y**. Our sequence now looks like this:

zydzyac

We have seven characters in our sequence, but our vocabulary isn't just four or five—it's now six. For the final round, we once again examine the sequence and find that the pair **zy** is most common. We replace it one more time with another character—let's call it **x**. So, **x** represents **zy**, and we replace all occurrences of **zy**. Our final sequence looks like this:

XdXac

So, basically, after we have gone through this process, instead of having a sequence of **11** tokens with a vocabulary length of **four**, we now have a sequence of **5** tokens. But our vocabulary length has increased to **seven**. In this way, we can iteratively compress our sequence. We mint new tokens. So, in the exact same way, we start out with byte sequences—our vocabulary size is **256**. However, we're now going to go through these and find the byte pairs that occur most frequently. We'll iteratively start minting new tokens, appending them to our vocabulary, and replacing things. This process allows us to end up with a compressed training dataset and an algorithm for taking any arbitrary sequence and encoding it using this vocabulary. We can also decode it back to strings.

The implementation

Let's now implement all that. Here's what I did: I went to this blog post that I enjoyed, and I took the first paragraph. I copied and pasted it here into text, resulting in one very long line. Now, to get the tokens, as I mentioned, we simply take our text and encode it into UTF-8. The tokens here, at this point, will be raw bytes—a single stream of bytes. However, to make it easier to work with, I'm going to convert all those bytes to integers and then create a list. This way, it's easier for us to manipulate and work with in Python, and we can visualize it. Here, I'm printing all of that.

So, this is the original paragraph, which has a length of **533** code points. Next, we have the bytes encoded in UTF-8, and we see that this has a length of **616** bytes (or 616 tokens). The reason it's more is that a lot of these simple ASCII characters or basic characters become a single byte. However, many Unicode complex characters consist of multiple bytes—up to four. So, we're expanding the size.

Now, as a first step in the algorithm, we'd like to iterate over this and find the pair of bytes that occur most frequently. We'll merge them. If you're working along in a notebook or on the side, I encourage you to click on the link, find this notebook, and try to write that function yourself.

Counting consecutive pairs, finding most common pair

Implement first the function that finds the most common pair. Okay, so here's what I came up with. There are many different ways to implement this, but I'm calling the function `get_stats`. It expects a list of integers. I'm using a dictionary to keep track of the counts. Additionally, this is a Pythonic way to iterate over consecutive elements of this list (which we covered in the previous video). For each pair, I increment the count by one. If I call this on all the tokens here, the stats come out as follows:

```
In [99]: def get_stats(ids):
          counts = {}
          for pair in zip(ids, ids[1:]): # Pythonic way to iterate consecutive elements
              counts[pair] = counts.get(pair, 0) + 1
          return counts

          stats = get_stats(tokens)
          # print(stats)
          print(sorted([(v,k) for k,v in stats.items()], reverse=True))

[(20, (101, 32)), (15, (240, 159)), (12, (226, 128)), (12, (105, 110)), (10, (115, 32)), (10, (97, 110)), (10, (32, 97)), (9, (32, 116)), (8, (116, 104)), (7, (159, 135)), (7, (159, 133)), (7, (97, 114)), (6, (239, 189)), (6, (140, 240)), (6, (128, 140)), (6, (116, 32)), (6, (114, 32)), (6, (111, 114)), (6, (110, 103)), (6, (110, 100)), (6, (109, 101)), (6, (104, 101)), (6, (101, 114)), (6, (32, 105)), (5, (117, 115)), (5, (115, 116)), (5, (110, 32)), (5, (100, 101)), (5, (44, 32)), (5, (32, 115)), (4, (116, 105)), (4, (116, 101)), (4, (115, 44)), (4, (114, 105)), (4, (111, 117)), (4, (111, 100)), (4, (110, 116)), (4, (110, 105)), (4, (105, 99)), (4, (104, 97)), (4, (103, 32)), (4, (101, 97)), (4, (100, 32)), (4, (99, 111)), (4, (97, 109)), (4, (85, 110)), (4, (32, 119)), (4, (32, 111)), (4, (32, 102)), (4, (32, 85)), (3, (118, 101)), (3, (116, 115)), (3, (116, 114)), (3, (116, 111)), (3, (114, 116)), (3, (
```

This is the dictionary. The keys are tuples of consecutive elements, and value is the count. To print it in a slightly better way, I use a compound approach. We iterate over all the items. The `items` method called on the dictionary returns pairs of key-value. Instead, I create a list here of value-key because if it's a value-key list, then I can call `sort` on it. By default, Python will use the first element (which, in this case, will be the value) to sort by. If it's given tuples, it sorts in descending order. So, when we print that, it looks like **101, 32** was the most commonly

occurring consecutive pair, and it occurred **20 times**. We can double-check that this makes reasonable sense. If I search for **101, 32**, you'll see that these are the 20 occurrences of that pair. And if we'd like to take a closer look at what exactly that pair is, we can use `chr()`, which is the opposite of `ord()` in Python. So, we give it a Unicode code point: **101** and **32**. We see that this corresponds to **'e'** and **space**. Essentially, there's a lot of **'e'** followed by space here, meaning that many of these words seem to end with **'e'**.

Merging the most common pair

So, here's an example: there's a lot of that going on here, and this is the most common pair. Now that we've identified the most common pair, we'd like to iterate over this sequence. We're going to mint a new token with the ID of **256**. Why **256**? Because these tokens currently go from **2** to **255**. When we create a new token, it will have an ID of **256**. We'll iterate over this entire list, and every time we encounter **101, 32**, we're going to swap that out for **256**. Let's implement that now. Feel free to do it yourself as well.

```
In [110]: def get_stats(ids):
          counts = {}
          for pair in zip(ids, ids[1:]): # Pythonic way to iterate consecutive elements
              counts[pair] = counts.get(pair, 0) + 1
          return counts

          stats = get_stats(tokens)
          # print(stats)
          # print(sorted(((v,k) for k,v in stats.items()), reverse=True))

In [111]: top_pair = max(stats, key=stats.get)
          top_pair

Out[111]: (101, 32)
```

First, I've commented out some code so that we don't clutter the notebook too much. This is a nice way in Python to obtain the highest-ranking pair. Essentially, we're calling `max` on this dictionary, `stats`, and it will return the maximum key. Now, how does it rank keys? You can provide it with a function that ranks keys. In this case, the function is just `stats.get`. `stats.get` will return the value, so we're ranking by the value and getting the maximum key. As we saw, it's **101, 32**.

```
In [114]: def merge(ids, pair, idx):
          # in the list of ints (ids), replace all consecutive occurrences of pair with the new token
          newids = []
          i = 0
          while i < len(ids):
              # if we are not at the very last position AND the pair matches, replace it
              if i < len(ids) - 1 and ids[i] == pair[0] and ids[i+1] == pair[1]:
                  newids.append(idx)
                  i += 2
              else:
                  newids.append(ids[i])
                  i += 1
          return newids

          print(merge([5, 6, 6, 7, 9, 1], (6, 7), 99))

          #tokens2 = merge(tokens, top_pair, 256)
          #print(tokens2)
          #print("length:", len(tokens2))

          [5, 6, 99, 9, 1]
```

Now, to actually merge **101,32**, this is the function I wrote. Again, there are many different versions of it. We're going to take a list of IDs and the pair that we want to replace. That pair will

be replaced with the new index, **idx**. We iterate through the IDs, and if we find the pair, we swap it out for **idx**. We create this new list, start at zero, and then go through the entire list sequentially from left to right. Here, we're checking for equality at the current position with the pair. Essentially, we're ensuring that the pair matches. Now, here's a bit of a tricky condition that you have to append.

If you're trying to be careful, here's the thing: you don't want `ids[i+1] == pair [1]`: to be out of bounds at the very last position when you're on the rightmost element of this list. Otherwise, it would give you an out-of-bounds error. So, we have to make sure that we're not at the very, very last element. Thus, this condition would be false for that case. If we find a match, we append the replacement index to this new list, and we increment the position by two. This way, we skip over that entire pair. However, if we haven't found a matching pair, we simply copy over the element at that position and increment by one. Then, we return this.

Here's a very small toy example: if we have a list **566, 791**, and we want to replace the occurrences of **67** with **99**, calling this function on that will give us the desired result. So, the **67** is replaced with **99**.

Now, I'm going to uncomment this for our actual use case. We want to take our tokens, take the top pair 101, 32 and replace it with **256** to get **tokens2**. If we run this, we get the following:

```
[5, 6, 99, 9, 1]
[239, 188, 181, 239, 189, 142, 239, 189, 137, 239, 189, 131, 239, 189, 143, 239, 189, 132, 239, 189, 133, 33, 32, 2
40, 159, 133, 164, 240, 159, 133, 157, 240, 159, 133, 152, 240, 159, 133, 146, 240, 159, 133, 158, 240, 159, 133, 1
47, 240, 159, 133, 148, 226, 128, 189, 32, 240, 159, 135, 186, 226, 128, 140, 240, 159, 135, 179, 226, 128, 140, 24
0, 159, 135, 174, 226, 128, 140, 240, 159, 135, 168, 226, 128, 140, 240, 159, 135, 180, 226, 128, 140, 240, 159, 13
5, 169, 226, 128, 140, 240, 159, 135, 170, 33, 32, 240, 159, 152, 132, 32, 84, 104, 256, 118, 101, 114, 121, 32, 11
0, 97, 109, 256, 115, 116, 114, 105, 107, 101, 115, 32, 102, 101, 97, 114, 32, 97, 110, 100, 32, 97, 119, 256, 105,
110, 116, 111, 32, 116, 104, 256, 104, 101, 97, 114, 116, 115, 32, 111, 102, 32, 112, 114, 111, 103, 114, 97, 109,
109, 101, 114, 115, 32, 119, 111, 114, 108, 100, 119, 105, 100, 101, 46, 32, 87, 256, 97, 108, 108, 32, 107, 110, 1
11, 119, 32, 119, 256, 111, 117, 103, 104, 116, 32, 116, 111, 32, 226, 128, 156, 115, 117, 112, 112, 111, 114, 116,
32, 85, 110, 105, 99, 111, 100, 101, 226, 128, 157, 32, 105, 110, 32, 111, 117, 114, 32, 115, 111, 102, 116, 119, 9
7, 114, 256, 40, 119, 104, 97, 116, 101, 118, 101, 114, 32, 116, 104, 97, 116, 32, 109, 101, 97, 110, 115, 226, 12
8, 148, 108, 105, 107, 256, 117, 115, 105, 110, 103, 32, 119, 99, 104, 97, 114, 95, 116, 32, 102, 111, 114, 32, 97,
108, 108, 32, 116, 104, 256, 115, 116, 114, 105, 110, 103, 115, 44, 32, 114, 105, 103, 104, 116, 63, 41, 46, 32, 6
6, 117, 116, 32, 85, 110, 105, 99, 111, 100, 256, 99, 97, 110, 32, 98, 256, 97, 98, 115, 116, 114, 117, 115, 101, 4
4, 32, 97, 110, 100, 32, 100, 105, 118, 105, 110, 103, 32, 105, 110, 116, 111, 32, 116, 104, 256, 116, 104, 111, 11
7, 115, 97, 110, 100, 45, 112, 97, 103, 256, 85, 110, 105, 99, 111, 100, 256, 83, 116, 97, 110, 100, 97, 114, 100,
32, 112, 108, 117, 115, 32, 105, 116, 115, 32, 100, 111, 122, 101, 110, 115, 32, 111, 102, 32, 115, 117, 112, 112,
100, 101, 100, 101, 110, 116, 07, 114, 101, 32, 07, 110, 110, 101, 100, 101, 115, 44, 32, 114, 101, 117, 111, 114
```

Recall that previously, we had a length of **616** in this list, and now we have a length of **596**. This decrease by **20** makes sense because there are **20** occurrences. Moreover, let's try to find **256** here, and we see plenty of occurrences of it. Furthermore, just to double-check, there should be no occurrence of **101, 32**. So, in the original array, there are plenty of them, but in the second array, there are no occurrences of **101, 32**. We've successfully merged 101, 32 into 256, and now we just iterate. We're going to go over the sequence again, find the most common pair, and replace it. Let me now write a **while loop** that uses these functions to do this iteratively. How many times do we do it? Well, that's totally up to us as a hyperparameter. The more steps we take, the larger our vocabulary will be and the shorter our sequence. There's a sweet spot that we usually find works best in practice. So, this is kind of a hyperparameter, and we tune it to find good vocabulary sizes. As an example, GPT-4 currently uses roughly **100,000** tokens. These are reasonable numbers for large language models. Now, let me put it all together and iterate through these steps.

Training the tokenizer: adding the while loop, compression ratio

Okay, now before we dive into the while loop, I wanted to add one more cell here. Instead of grabbing just the first paragraph or two from the blog post, I took the entire blog post and stretched it out into a single line. Basically, using longer text will allow us to have more representative statistics for the byte pairs, and we'll get more sensible results because it's longer text.

```
# making the training text longer to have more representative token statistics
# text from https://www.reedbeta.com/blog/programmers-intro-to-unicode/
text = """"A Programmer's Introduction to Unicode March 3, 2017 · Coding · 22 Comments Unicode! UNICODE? U\
tokens = text.encode('utf-8') # raw bytes
tokens = list(map(int, tokens)) # convert to a list of integers in range 0..255 for convenience
```

So, here we have the raw text. We encode it into bytes using the UTF-8 encoding. As before, we convert it into a list of integers in Python, making it easier to work with instead of the raw byte objects. These two functions here are identical to what we had above; I included them here just for your reference. Now, let's discuss the new code I added.

```
In [31]: def get_stats(ids):
          counts = {}
          for pair in zip(ids, ids[1:]):
              counts[pair] = counts.get(pair, 0) + 1
          return counts

          def merge(ids, pair, idx):
              newids = []
              i = 0
              while i < len(ids):
                  if i < len(ids) - 1 and ids[i] == pair[0] and ids[i+1] == pair[1]:
                      newids.append(idx)
                      i += 2
                  else:
                      newids.append(ids[i])
                      i += 1
              return newids

          # ---
          vocab_size = 276 # the desired final vocabulary size
          num_merges = vocab_size - 256
          ids = list(tokens) # copy so we don't destroy the original list

          merges = {} # (int, int) -> int
          for i in range(num_merges):
              stats = get_stats(ids)
              pair = max(stats, key=stats.get)
              idx = 256 + i
              print(f"merging {pair} into a new token {idx}")
              ids = merge(ids, pair, idx)
              merges[pair] = idx
```

The first thing we want to do is decide on the final vocabulary size for our tokenizer. This is a hyperparameter, and you set it based on your best performance. For our case, let's use 276. That way, we'll be doing exactly 20 merges. Why 20? Because we already have 256 tokens for the raw bytes, and to reach 276, we need to add 20 new tokens. In Python, you can create a copy of a list using this approach."

ids = list(tokens)

So, I'm taking the tokens list and by wrapping it in a list, Python will construct a new list of all the individual elements. This is just a copy operation.

Then, **merges = {}**, this creates a merges dictionary. This merges dictionary is going to maintain the child-one-to-child-two mapping to a new token. What we're building up here is a binary tree of merges. However, it's not exactly a tree because a tree would have a single root node with a bunch of leaves. For us, we're starting with the leaves on the bottom, which are the individual bytes—those are the starting 256 tokens. Then, we're merging two of them at a time, so it's more like a forest. As we merge these elements, for 20 merges, we're going to find the most commonly occurring pair. We'll mint a new token integer for it. Here, **for i in range (num_merges)**: starts at zero, so we'll begin at 256. We'll print that we're merging it, and we'll

replace all occurrences of that pair with the new, newly minted token. We'll record that this pair of integers merged into this new integer. Running this gives us the following output:

```
merging (101, 32) into a new token 256
merging (105, 110) into a new token 257
merging (115, 32) into a new token 258
merging (116, 104) into a new token 259
merging (101, 114) into a new token 260
merging (99, 111) into a new token 261
merging (116, 32) into a new token 262
merging (226, 128) into a new token 263
merging (44, 32) into a new token 264
merging (97, 110) into a new token 265
merging (111, 114) into a new token 266
merging (100, 32) into a new token 267
merging (97, 114) into a new token 268
merging (101, 110) into a new token 269
```

We did 20 merges, and for example, the first merge was exactly as before—the 101,32 tokens merging into a new token, 256. Keep in mind that the individual tokens 101 and 32 can still occur in the sequence after merging; it's only when they occur exactly consecutively that they become 256. Also, notice that the token 256, which is the newly minted token, is also eligible for merging. On the bottom, **merging (259, 256) into new token 275**, the 20th merge was a merge of 25 and 259, becoming 275. So, every time we replace these tokens, they become eligible for merging in the next round of data iteration. That's why we're building up a small sort of binary forest instead of a single individual tree. One thing we can take a look at is the compression ratio that we've achieved.

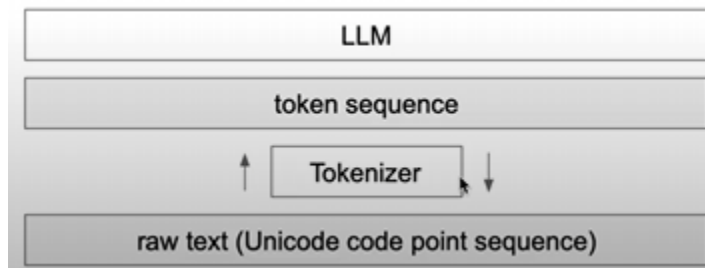
```
In [32]: print("tokens length:", len(tokens))
          print("ids length:", len(ids))
          print(f"compression ratio: {len(tokens) / len(ids):.2f}X")

tokens length: 24597
ids length: 19438
compression ratio: 1.27X
```

In particular, we started off with this tokens list—24,000 bytes. After merging 20 times, we now have only 19,000 tokens. Therefore, the compression ratio, simply dividing the two, is roughly 1.27. That's the amount of compression we were able to achieve for this text with only 20 merges. Of course, the more vocabulary elements you add, the greater the compression ratio would be. Finally, that's kind of like the training of the tokenizer, if you will.

Tokenizer/LLM diagram: it is a completely separate stage

Now, one point I wanted to make is that—maybe this diagram can help



—the tokenizer is a completely separate object from the large language model itself. So, everything in this lecture doesn't really touch the LLM itself; we're just training the tokenizer. This is a completely separate pre-processing stage. Usually, the tokenizer will have its own training set, just like a large language model has a potentially different training set. The tokenizer has a training set of documents on which you're going to train it. Then, we're performing the byte pair encoding algorithm, as we saw above, to train the vocabulary of this tokenizer. It has its own training set. It's a pre-processing stage that you would run only once at the beginning. Once you have the tokenizer trained and you have the vocabulary and the merges, we can do both encoding and decoding. The two arrows represent the tokenizer as a translation layer between raw text (which is, as we saw, the sequence of Unicode code points). It can take raw text and turn it into a token sequence and vice versa, it can take a token sequence and translate it back into raw text. Now that we have the trained tokenizer and these merges, we'll turn to how we can perform the encoding and decoding steps. If you give me text, I'll provide the tokens, and vice versa—if you give me tokens, I'll generate the corresponding text. Once we have that, we can seamlessly translate between these two realms. Afterward, the language model will be trained as the next step. Typically, in a state-of-the-art application, you might take all of your training data for the language model and run it through the tokenizer, translating everything into a massive token sequence. You might run it through the tokenizer and sort of translate everything into a massive token sequence. Then, you can throw away the raw text—you're just left with the tokens themselves. Those tokens are stored on disk, and that is what the large language model is actually reading when it's training on them.

So, this is one approach that you can take as a single, massive pre-processing step. It's a separate stage, and it usually has its own entire training set. You may want to have those training sets be different between the tokenizer and the large language model. For example, when you're training the tokenizer, we don't just care about the performance of English text. We also care about many different languages and whether the text contains code. Therefore, you might want to explore different mixtures of languages and varying amounts of code in your training data. The diversity of languages in your tokenizer training set will **determine how many merges occur**, which in turn affects the density of this type of data in the token space.

Basically, if you add a substantial amount of Japanese data to your tokenizer training set, more Japanese tokens will get merged. Consequently, Japanese sequences will be shorter. This can be beneficial for the large language model, which has a finite context length in the token space.

Now, let's delve into encoding and decoding since we've trained a tokenizer and have our merges. How do we perform encoding and decoding?

Decoding tokens to strings

So, let's begin with decoding, which is the right arrow from the diagram. Given a token sequence, let's go through the tokenizer to get back a Python string object—the raw text. This is the function we'd like to implement. We're given a list of integers, and we want to return a Python string.

```
vocab = {idx: bytes([idx]) for idx in range(256)}
for (p0, p1), idx in merges.items():
    vocab[idx] = vocab[p0] + vocab[p1]

def decode(ids):
    # given ids (list of integers), return Python string
    pass
```

There are many different ways to do it. Here's one approach: I'll create a preprocessing variable called vocab. Vocab is a mapping or dictionary in Python from the token ID to the bytes object for that token. We begin with the raw bytes for tokens from 0 to 255. Then, we go in order of all the merges, populating this vocab list by concatenating the bytes representation of the first child followed by the second one. **vocab[idx] = vocab[p0] + vocab[p1]** Remember, these are bytes objects, so this addition is just concatenation.

One tricky thing to be careful with, by the way, is that I'm iterating through a dictionary in Python using **.items()**. It really matters that this runs in the order in which we inserted items into the merges dictionary. Luckily, starting with Python 3.7, this is guaranteed to be the case. But before Python 3.7, this iteration may have been out of order with respect to how we inserted elements into merges, and this may not have worked. However, we are using modern Python, so we're okay. Now, given the IDs, the first thing we're going to do is get the tokens.

```
def decode(ids):
    # given ids (list of integers), return Python string
    tokens = b"".join(vocab[idx] for idx in ids)
    text = tokens.decode("utf-8")
    pass
```

The way I implemented this here is by iterating over all the IDs. I'm using vocab to look up their bytes. Then, this is one way in Python to concatenate all these bytes together to create our tokens. At this point, these tokens are raw bytes, so I have to decode them using UTF-8 back into Python strings. Previously, we called encode on a string object to get the bytes, and now we're doing the opposite. We're taking the bytes and calling decode on the bytes object to get a string in Python. Finally, we can **return text**.

However, this implementation has an issue that could actually throw an error. Try to figure out why this code might result in an error if we plug in some sequence of IDs that is unlucky. Let me demonstrate the issue:

```
print(decode([97]))
a
```

When I try to decode just something like 97, I get the letter ‘a’ here back—nothing too crazy happening. But when I try to decode 128 as a single element, the token 128 results in an error. In string or in Python object, UnicodeDecodeError: ‘utf-8’ can’t decode byte 0x80 (which is this in HEX) in position 0: invalid start byte.

What does that mean? Well, to understand, we have to go back to our UTF-8 page that I briefly showed earlier. And this is Wikipedia UTF-8. Basically, there’s a specific schema that UTF-8 bytes follow.

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	11xxxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

In particular, if you have a multi-byte object for some of the Unicode characters, they have to have this special sort of envelope in how the encoding works. So, what’s happening here is that we encounter an ‘invalid start byte.’ That’s because 128, the binary representation of it, is one followed by all zeros. We see here that it doesn’t conform to the format because one followed by all zeros just doesn’t fit any of these rules. So, it’s an invalid start byte, which is byte one. This byte must have a one following it and then a zero following it, and then the content of your Unicode character.

Essentially, we don’t exactly follow the UTF-8 standard, and this cannot be decoded. The way to fix this is to use the **errors='replace'** argument in Python’s **bytes.decode()** function. By default, errors is set to ‘strict,’ so it throws an error if it encounters invalid UTF-8 bytes encoding. However, you can choose different error handling options. The full list of errors you can use is available. In particular, let’s change it to ‘replace,’ and that will replace invalid bytes with a special marker—the replacement character.

```
vocab = {idx: bytes([idx]) for idx in range(256)}
for (p0, p1), idx in merges.items():
    vocab[idx] = vocab[p0] + vocab[p1]

def decode(ids):
    # given ids (list of integers), return Python string
    tokens = b"".join(vocab[idx] for idx in ids)
    text = tokens.decode("utf-8", errors="replace")
    return text

print(decode([128]))
 
```

Now, we just get that character back. So, not every single byte sequence is valid UTF-8. If your large language model, for example, predicts tokens in a bad manner, they might not fall into valid UTF-8, and then we won’t be able to decode them. The standard practice is to use **errors='replace,'** and you’ll find this in the OpenAI code as well. Basically, whenever you see this

kind of character in your output, something went wrong, and the LM output was not a valid sequence of tokens.

Encoding strings to tokens

Now we're going to go the other way. We're going to implement the left arrow. We'll be given a string, and we want to encode it into tokens. So, this is the signature of the function we're interested in. This function should basically print a list of integers representing the tokens.

```
def encode(text):  
    # given a string, return list of integers (the tokens)  
    tokens = list(text.encode("utf-8"))  
    while True:  
        stats = get_stats(tokens)  
        pair = min(stats, key=lambda p: merges.get(p, float("inf")))  
  
    print(encode("hello world!"))
```

There are many ways to do this, but here's one approach I came up with. First, we'll take our text and encode it into UTF-8 to get the raw bytes. Then, as before, we'll call `list()` on the bytes object to get a list of integers representing those bytes. These are the starting tokens—the raw bytes of our sequence.

Now, according to the merges dictionary above (recall that this was the merges), some of the bytes may be merged based on this lookup. Additionally, remember that the merges were built from top to bottom, so we prefer to do all these merges in the beginning before we do any later merges. For example,

(259, 256): 275}

the merge over here relies on the 256, which got merged earlier. Therefore, we need to follow the order from top to bottom when merging anything.

Now, we expect to perform a few merges, so we add **while True:**. Let's find a pair of bytes that is consecutive and allowed to merge according to this lookup. To reuse some of the functionality we've already written, I'll use the **get_stats** function. Recall that **get_stats** counts how many times every single pair occurs in our sequence of tokens and returns that information as a dictionary.

Many times, every single pair occurs in our sequence of tokens and returns that as a dictionary. The dictionary represents a mapping from all the different byte pairs to the number of times they occur. At this point, we don't actually care how many times they occur in the sequence; we only care about what the raw pairs are in that sequence. So, I'm only going to be using the keys of the dictionary. I'm interested in the set of possible merge candidates, if that makes sense.

Now, we want to identify the pair that we're going to be merging at this stage of the loop. What do we want? We want to find the pair (or key) inside stats that has the lowest index in the merges dictionary. Why? Because we want to perform all the early merges before working our way to the late merges.

```
pair = min(stats, key=lambda p: merges.get(p, float("inf")))
```

To achieve this, I'll use a little trick. I'll employ the **min** function over an iterator in Python. When you call **min** on an iterator (in this case, **stats** as a dictionary), we'll iterate through the keys of this dictionary. Specifically, we're looking at all the pairs inside **stats**—which are all the consecutive pairs. We'll find the consecutive pair inside **tokens** that has the minimum index. The **min** function takes a key function, which gives us the function that will return a value over which we'll perform the minimum operation. In our case, we care about taking **merges** and getting the index of that pair. So, for any pair inside **stats**, we'll look into **merges** to find its index and return the pair with the minimum number.

For example, if there's a pair 101 and 32, we definitely want to identify it here and return it. The resulting pair would become 101, 32 if it occurs.

And the reason I'm putting a **float INF** here as a fallback is that in the **get()** function, when we basically consider a pair that doesn't occur in the **merges**, that pair is not eligible to be merged. So, if in the token sequence there's some pair that is not a merging pair (meaning it cannot be merged), it doesn't actually occur here. It also doesn't have an index, and it cannot be merged. We'll denote this as **float INF**. The reason **Infinity** is nice here is that we're guaranteed it won't participate in the list of candidates when we do the minimum operation. So, this is one way to handle it.

In short, this function returns the most eligible merging candidate pair that occurs in the **tokens**. However, there's one thing to be careful with: this function might fail if there's nothing left to merge. If there's nothing in **merges** that satisfies the merging criterion anymore, everything just returns **float INF**. In that case, the pair will simply become the very first element of **stats**. But this pair is not actually a mergeable pair; it's just the first pair inside **stats** arbitrarily because all of these pairs evaluate to **float INF** for the merging criterion.

```
def encode(text):  
    # given a string, return list of integers (the tokens)  
    tokens = list(text.encode("utf-8"))  
    while True:  
        stats = get_stats(tokens)  
        pair = min(stats, key=lambda p: merges.get(p, float("inf")))  
        if pair not in merges:  
            break # nothing else can be merged  
    |
```

So, if this pair is not in the returned **merges**, it signals that there is nothing left to merge—no single pair can be merged anymore. In that case, we'll break out, and nothing else can be merged. You might come up with a different implementation, but this approach ensures we find a pair that can be merged with the lowest index.

```
def encode(text):
    # given a string, return list of integers (the tokens)
    tokens = list(text.encode("utf-8"))
    while True:
        stats = get_stats(tokens)
        pair = min(stats, key=lambda p: merges.get(p, float("inf")))
        if pair not in merges:
            break # nothing else can be merged
        idx = merges[pair]
        tokens = merge(tokens, pair, idx)
    return tokens

print(encode("hello world!"))
```

Now, if we did find a pair inside merges with the lowest index, we can merge it. So we're going to look into the merger dictionary for that pair. We'll look up the index, and now we're going to merge that into that index. So, we're going to do **tokens = merge()** the original tokens. We're going to be replacing the pair 'pair,' and we'll replace it with 'index' (abbreviated as 'idx'). This returns a new list of tokens where every occurrence of 'pair' is replaced with 'idx.' We're doing a merge, and we'll continue this process until eventually nothing can be merged. When we reach that point, we'll break out, and at the end we'll just return the tokens. So, that's the implementation.

```
print(encode("hello world!"))
[104, 101, 108, 108, 111, 32, 119, 266, 108, 100, 33]
```

This looks reasonable. For example, 32 represents a space in ASCII, so that's here. It seems like it worked great. Now, let's wrap up this section of the video. I want to point out that this is not quite the right implementation just yet because we're leaving out a special case. Specifically, if we try to do **print(encode("h"))**, it would give us an error. The issue is that if we only have a single character or an empty string, then stats is empty, causing an issue inside **Min**. To address this, one way is to check if the length of tokens (denoted as L) is at least two **while len(tokens) >= 2**. If it's less than two, meaning it's just a single token or no tokens, then there's nothing to merge, so we simply return. That should fix that case. Additionally, I have a few test cases here for us. First, let's make sure that if we take a string, encode it, and then decode it back, we'd expect to get the same string back. Is that true for all strings?

```
print(decode(encode("hello world")))
hello world
```

So, I think here it is the case, and I believe, in general, this is probably true. However, notice that going backwards is not an identity. Going backwards is not guaranteed because, as I mentioned, not all token sequences are valid UTF-8. Some of them can't even be decoded. So, this process only goes in one direction. But for that one direction, we can check. If we take the training text (which is the text we use to train the tokenizer),

```
text2 = decode(encode(text))
print(text2 == text)

True
```

we can ensure that when we encode and decode, we get the same thing back, which is true. Additionally, I took some validation data from a web page.

```
valtext = "Many common characters, including numerals, punctuation, and other symbols"
valtext2 = decode(encode(valtext))
print(valtext2 == valtext)
```

True

This text is unseen by the tokenizer, and we can verify that it also works. Okay, this gives us confidence that the implementation is correct. So, those are the basics of the Byte Pair Encoding algorithm. We saw how we can take a training set, train a tokenizer, and create the parameters for this tokenizer.

```
{(101, 32): 256,
 (105, 110): 257,
 (115, 32): 258,
 (116, 104): 259,
 (101, 114): 260,
 (99, 111): 261,
 (116, 32): 262,
 (226, 128): 263,
 (44, 32): 264,
 (97, 110): 265,
 (111, 114): 266,
 (100, 32): 267,
 (97, 114): 268,
 (101, 110): 269,
 (257, 103): 270,
 (261, 100): 271,
 (121, 32): 272,
 (46, 32): 273,
 (97, 108): 274,
 (259, 256): 275}
```

The parameters are essentially just this dictionary of merges, which creates a binary forest on top of raw bytes. Once we have this merges table, we can both encode and decode between raw text and token sequences. That's the simplest setting of the tokenizer. Now, let's explore some state-of-the-art language models and the tokenizers they use. We'll see that this picture becomes more complex, so we'll delve into the details of this complexity step by step..

Regex patterns to force splits across categories

So, let's kick things off by looking at the **GPT-2 Series**. In particular, I have the GPT-2 paper here. This paper is from 2019, so about 5 years ago. Let's scroll down to the section on input representation. This is where they discuss the tokenizer used for GPT-2.

Now, this section is fairly readable, so I encourage you to pause and read it yourself. Here, they motivate the use of the **byte pair encoding (BPE) algorithm** on the byte-level representation of **UTF-8 encoding**. They talk about vocabulary sizes and other relevant details.

However, things start to diverge from what we've covered so far. They mention that they don't simply apply the naive algorithm as we have done. Specifically, consider the example of common words like 'dog'. In text, 'dog' frequently occurs right next to various punctuation marks, such as 'dog.', 'dog!', and 'dog?'. Naively, the BPE algorithm might merge these into a single token, resulting in many tokens that are essentially just variations of 'dog' with slightly different punctuation. This clustering seems suboptimal, especially when combining semantics with punctuation.

To address this, they take a **top-down manual approach** to enforce rules that prevent certain types of characters from being merged together. Essentially, they fine-tune the BPE algorithm to handle specific cases more effectively.

If you're curious, you can explore their code on the **GPT-2 repository on GitHub**¹. And when we go to the source, there is an encoder. Now, I don't personally love that they call it 'encoder' because this is the tokenizer, and the tokenizer can both encode and decode. So, it feels kind of awkward to me that it's called an encoder, but that is indeed the tokenizer. There's a lot going on here, and we're going to step through it in detail. For now, I want to focus on this part: **'create a regex pattern'**. This pattern looks very complicated, and we'll dive into it shortly.

However, this core part allows them to enforce rules for which parts of the text will never be merged. Now, notice that **re.compile** here is a little bit misleading. We're not just doing **import re**, which is the Python **re** module. Instead, we're doing **import reex as re**. **reex** is a Python package that you can install with **pip install rx**. It's essentially an extension of **re**, making it more powerful.

```
import regex as re
gpt2pat = re.compile(r'""s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?(?:\s\p{L}\p{N})+|\s+(?!S)|\s+""')
print(re.findall(gpt2pat, "Hello world"))
```

Let's take a closer look at this pattern and understand why it achieves the desired separation. I've copy-pasted the pattern into our Jupyter notebook where we left off. Now, just like their code, we'll call **re.findall** for this pattern on any arbitrary string that we're interested in. This string is what we want to encode into tokens to feed into NLLM (like GPT-2).

So, what exactly is this pattern doing? **re.findall** will take this pattern and try to match it against a string. The process involves moving from left to right in the string, attempting to match the pattern. **re.findall** will collect all occurrences and organize them into a list.

When you examine the pattern, first notice that `"r"""""` is a raw string. The three double quotes are just used to start the string. So, really, the string itself—this is the pattern itself, right? Notice that it's made up of a lot of **'or'** operators (vertical bars) in **regex**. When you go from left to right in this pattern and try to match it against the string, wherever you are, we encounter **'hello'**. Let's analyze it further.

It's not **'apostrophe s'** or **'apostrophe t'**, but it is an optional space followed by `\p{L}+` (one or more times). What is `\p{L}+`? According to some documentation I found (there might be other sources as well), `\p{L}+` represents any kind of letter from any language. Since **'hello'** is made up of letters **h, e, l**, etc., the optional space followed by a bunch of letters (one or more) matches **'hello'**. However, the match ends because a white space is not a letter. From there, a new attempt to match against the string begins. We skip over all these elements again until we reach the exact same point. Now, we encounter another optional space followed by a bunch of letters (one or more). This matches the space between **'hello'** and **'world'**. So, when we run this, we get a list of two elements: **'hello'** and then **'world'**.

But what is this doing, and why is it important? Instead of directly encoding the string for tokenization, we first split it up. When you step through the code, you'll see that it splits your text into a list of texts. Each element of this list is processed independently by the tokenizer, and all the results are simply concatenated. So, **'hello world, how are you?'** becomes five elements in the list. All of these will independently go from text to a token sequence, and then that token sequence is going to be concatenated. It's all going to be joined up, and roughly speaking, what that does is you're only ever finding merges between the elements of this list. So, you can only ever consider merges within every one of these elements individually. After you've done all the possible merging for all of these elements individually, the results of all that will be joined by concatenation.

```
import regex as re
gpt2pat = re.compile(r""""s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+ ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+""")
print(re.findall(gpt2pat, "Hello world how are you"))
['Hello', ' world', ' how', ' are', ' you']
```

Essentially, you are never going to be merging **'e'** **"are"** with this space **"space you"** because they are now parts of the separate elements of this list. Therefore, you are saying we are never going to merge **'e'** and **'space'** because we're breaking it up in this way. Basically, using this regex pattern to chunk up the text is just one way of enforcing that some merges are not to happen. We're going to go into more of this text, and we'll see that what this is trying to do on a high level is to avoid merging across letters, numbers, punctuation, and so on. Let's see in more detail how that works.


```
import regex as re
gpt2pat = re.compile(r"''''s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\d{N}+| ?(?:\s\p{L}\p{N})+|\s+"
print(re.findall(gpt2pat, "Hello've world123 HOW'S are you"))
['Hello', "'ve", ' world', '123', ' HOW', "'", 'S', ' are', ' you']
```

So, let's continue. Now we have `'\p{n)'` If you go to the documentation, `'\p{n)'` is any kind of numeric character in any script. So, it's numbers. We have an optional space followed by numbers, and those would be separated out. Letters and numbers are being separated. For example, if I type 'Hello World 123, how are you?' then 'World' will stop matching here because '1' is not a letter anymore, but it's a number. This group will match for that, and we'll get it as a separate entity. Let's see how these apostrophes work. So, here, if we have "ve or, apostrophe 've' as an example, then the apostrophe here is not a letter or a number. So, 'Hello' will stop matching. ---That will come out separate thing.

All of these will independently go from text to a token sequence, and then that token sequence is going to be concatenated. It's all going to be joined up, and roughly speaking, what that does is you're only ever finding merges between the elements of this list. So, you can only ever consider merges within every one of these elements individually. After you've done all the possible merging for all of these elements individually, the results of all that will be joined by concatenation. Essentially, you are never going to be merging 'e' "are" with this space "space you" because they are now parts of the separate elements of this list. Therefore, you are saying we are never going to merge 'e and space' because we're breaking it up in this way. Basically, using this regex pattern to chunk up the text is just one way of enforcing that some merges are not to happen. We're going to go into more of this text, and we'll see that what this is trying to do on a high level is to avoid merging across letters, numbers, punctuation, and so on. Let's see in more detail how that works.

So, let's continue. Now we have `'\p{n)'` If you go to the documentation, `'\p{n)'` is any kind of numeric character in any script. So, it's numbers. We have an optional space followed by numbers, and those would be separated out. Letters and numbers are being separated. For example, if I type 'Hello World 123, how are you?' then 'World' will stop matching with "123" because '1' is not a letter anymore, but it's a number. `?\p{N}+` will match for that, and we'll get it as a separate entity. Let's see how these apostrophes work. So, here, if we have "ve' or, apostrophe 've' as an example, then the apostrophe here is not a letter or a number. So, 'Hello' will stop matching and then we will exactly match it with `| 've|`. so why are we using apostrophes before the letters? because they're the most common apostrophes used in the english language but this can cause some problems when using unicode apostrophes.

As a result, we try to match letters, then we try to match numbers. And then, if that doesn't work, we fall back `?(^s\p{L}\p{N})+`. What this is saying is an optional space followed by something that is not a letter, number, or a space in one or more of that. So, what `?(^s\p{L}\p{N})+` is doing effectively is trying to match punctuation, roughly speaking, not letters and not numbers. `?(^s\p{L}\p{N})+` will try to trigger for that. If I do something like

“!!!?”, then these parts here are not letters or numbers, but they will actually get caught here, and so they become their own group. We’ve separated out the punctuation.

Finally, `\s+(?!\\S)|\\s+` is also a little bit confusing. This is matching white space, but it’s using a **negative look-ahead assertion in regex**. What this is doing is matching white space up to but not including the last white space character. Why is this important? It’s pretty subtle, I think. You see how the white space is always included at the beginning of the word. So, ‘space are’ and ‘space you,’ etc. Suppose we have a lot of spaces here. What’s going to happen is that these spaces, up to but not including the last character, will get caught by this. And what that will do is separate out the spaces up to but not including the last character so that the last character can come here and join with the ‘space you.’ The reason that’s nice is because ‘space you’ is the common token. So, if I didn’t have these extra spaces here, you would just have ‘space you.’ And if I add tokens, if I add spaces, we still have a ‘space you,’ but now we have all this extra white space. Basically, the GPT-2 tokenizer really likes to have a space, letters, or numbers. It pre-pends these spaces, and this is just something that it is consistent about. So that’s what that is for.

And then finally, we have the last fallback: white space characters. So, if that doesn’t get caught, then this `\\s+` will catch any trailing spaces and so on.

```
example = """
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
"""
print(re.findall(gpt2pat, example))
```

[`'\n'`, `'for'`, `'i'`, `'in'`, `'range'`, `'('`, `'1'`, `' '`, `'101'`, `','`, `':'`, `'\n'`, `' '`, `'if'`, `'i'`, `' '`, `'and'`, `'i'`, `' '`, `'%'`, `'5'`, `' '`, `'=='`, `'0'`, `' '`, `':'`, `'\n'`, `' '`, `'print'`, `'('`, `'"`, `'FizzBuzz'`, `'"`, `' '`, `','`, `'\n'`, `' '`, `'3'`, `' '`, `'=='`, `'0'`, `' '`, `':'`, `'\n'`, `' '`, `'print'`, `'('`, `'"`, `'Fizz'`, `'"`, `' '`, `','`, `'\n'`, `' '`, `'elif'`, `'i'`, `' '`, `'%'`, `'5'`, `' '`, `'=='`, `'0'`, `' '`, `':'`, `'\n'`, `' '`, `'print'`, `'('`, `'"`, `'Buzz'`, `'"`, `' '`, `','`, `'\n'`, `' '`, `'else'`, `' '`, `':'`, `'\n'`, `' '`, `'print'`, `'('`, `'"`, `'i'`, `'"`, `' '`, `','`, `'\n'`, `'n']`

I wanted to show one more real-world example here. If we have this string, which is a piece of Python code, and then we try to split it up, then this is the kind of output we get. You’ll notice that the list has many elements here, and that’s because we are splitting up fairly often every time a category changes. So, there will never be any merges within these elements, and that’s what you are seeing here. Now, you might think that in order to train the tokenizer, OpenAI has used this to split up text into chunks and then run just a **BPE algorithm** within all the chunks. But that is not exactly what happened, and the reason is the following: notice that we have the **spaces** here before `print("fizzbuzz")`. Those spaces end up being entire elements, but these spaces never actually end up being merged by OpenAI. The way you can tell is that if you copy-paste the exact same chunk here into the tokenize function of the tokenizers library, you see that all the spaces are kept independent, and they’re all token 220. So, I think OpenAI, at some point, enforced some rule that these spaces would never be merged. Therefore, there are some additional rules on top of just chunking and BPE that OpenAI is not clear about. Now, the

training code for the GPT-2 tokenizer was never released, so all we have is the code that I've already shown you. But this code here that they've released is only the inference code for the tokens. So, this is not the training code. You can't give it a piece of text and train the tokenizer. This is just the inference code, which takes the merges that we already applies them to a new piece of text. So we don't know exactly how openai trained the tokenizer but it wasn't as simple as chunk it up and BPE it whatever it was.

Tiktoken library intro, differences between GPT-2/GPT-4 regex

```
import tiktoken

# GPT-2 (does not merge spaces)
enc = tiktoken.get_encoding("gpt2")
print(enc.encode("    hello world!!!"))

# GPT-4 (merges spaces)
enc = tiktoken.get_encoding("cl100k_base")
print(enc.encode("    hello world!!!"))

[220, 220, 220, 23748, 995, 10185]
[262, 24748, 1917, 12340]
```

Next, I wanted to introduce you to the tiktoken library from OpenAI, which is the official library for tokenization. To install it, use `pip install tiktoken`. You can do the tokenization in inference. This is again not training code; it's only inference code for tokenization. I wanted to show you how you would use it. Quite simple! Running this just gives us the GPT-2 tokens or the GPT-4 tokens. This is the tokenizer used for GPT-4. In particular, we see that the white space in GPT-2 remains unmerged, but in GPT-4, these white spaces merge. Now, in the GPT-4 tokenizer, they changed the regular expression that they use to chunk up text. The way to see this is by examining the tiktoken library. If you go to the file `tiktoken_ext/openai_public.py`, you'll find the definition of all these different tokenizers that OpenAI maintains.

```
"pat_str": r"^{(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?(?:^|\p{L}|\p{N})+| |\s+(?!\\S)|\s+}$",
"mergeable_ranks": mergeable_ranks,
"special_tokens": {ENDOFTEXT: 50256},
```

For GPT-2, the string is slightly different, but it's actually equivalent to what we discussed here.

So, this pattern that we discussed is equivalent to this pattern. This one just executes a little bit faster. Here, you see a slightly different definition, but otherwise, it's the same. We're going to go into special tokens in a bit. If you scroll down to CL 100k,

```
"name": "c1l00k_base",
"pat_str": "r====[?i:[sdmt][ll|ve|re][[^\r\n\\p{L}\\p{N}]?+\\p{L}+|\\p{N}{1,3}| ?[^\s\\p{L}\\p{N}]]+",
"mergeable_ranks": mergeable_ranks,
"special_tokens": special_tokens,
```

this is the GPT-4 tokenizer. You see that the pattern has changed. This is kind of like the main, major change, in addition to a bunch of other special tokens, which I'll go into in a bit again. Now, I'm not going to actually go into the full detail of the pattern change because, honestly, this is mind-numbing. I would just advise that you pull out ChatGPT and the regex documentation and step through it. But really, the major changes are: number one, you see this

'i' in ""(?i:". That means that the case-insensitive match. So, the comment that we saw earlier on, 'Oh, we should have used re.IGNORECASE,' basically, we're now going to be matching these apostrophes, 's', 'd', 'm,' etc. We're going to be matching them both in lowercase and in uppercase. So, that's fixed. There's a bunch of different handling of the white space that I'm not going to go into the full details of. And then one more thing here: you will notice that when they match the numbers, they only match one to three numbers. `"\p{N}{1,3}`

So, they will never merge numbers that are in more than three digits. Only up to three digits of numbers will ever be merged, and that's one change that they made as well to prevent tokens that are very, very long number sequences. But again, we don't really know why they do any of this stuff because none of this is documented, and it's just we get the pattern. So, yeah, it is what it is, but those are some of the changes that GPT-4 has made. And of course, the vocabulary size went from roughly 50k to roughly 100k.

```
✓ def get_encoder(model_name, models_dir):
    with open(os.path.join(models_dir, model_name, 'encoder.json'), 'r') as f:
        encoder = json.load(f)
    with open(os.path.join(models_dir, model_name, 'vocab.bpe'), 'r', encoding="utf-8") as f:
        bpe_data = f.read()
    bpe_merges = [tuple(merge_str.split()) for merge_str in bpe_data.split('\n')[1:-1]]
    return Encoder(
        encoder=encoder,
        bpe_merges=bpe_merges,
    )
```

The next thing I would like to do very briefly is to take you through the GPT-2 encoder, copy that. OpenAI has released this file, which I already mentioned to you briefly. Now, this file is fairly short and should be relatively understandable to you at this point. Starting at the bottom, they are loading two files: encoder.json and vocab.bpe. They do some light processing on it, and then they call this encoder object, which is the tokenizer. If you'd like to inspect these two files, which together constitute their saved tokenizer, you can do that with a piece of code like this.

```
# to download these two files:
#!wget https://openaipublic.blob.core.windows.net/gpt-2/models/1558M/vocab.bpe
#!wget https://openaipublic.blob.core.windows.net/gpt-2/models/1558M/encoder.json

import os, json

with open('encoder.json', 'r') as f:
    encoder = json.load(f)

with open('vocab.bpe', 'r', encoding="utf-8") as f:
    bpe_data = f.read()
bpe_merges = [tuple(merge_str.split()) for merge_str in bpe_data.split('\n')[1:-1]]
```

This is where you can download these two files and inspect them if you'd like. What you will find is that this encoder, as they call it in their code, is exactly equivalent to our vocab. So, remember here where we have this vocab object, which allowed us to decode very efficiently and basically took us from the integer to the bytes. Our vocab is exactly their encoder, and then their vocab.bpe, confusingly, is actually our merges. So, their BPE merges, which is based on the data inside vocab.bpe, ends up being equivalent to our merges. Basically, they are saving and loading the two variables that, for us, are also critical: the merges variable and the vocab

variable. Using just these two variables, you can represent a tokenizer, and you can both do encoding and decoding once you've trained this tokenizer.

```
class Encoder:
    def __init__(self, encoder, bpe_merges, errors='replace'):
        self.encoder = encoder
        self.decoder = {v:k for k,v in self.encoder.items()}
        self.errors = errors # how to handle errors in decoding
        self.byte_encoder = bytes_to_unicode()
        self.byte_decoder = {v:k for k, v in self.byte_encoder.items()}
        self.bpe_ranks = dict(zip(bpe_merges, range(len(bpe_merges))))
        self.cache = {}
```

Now, the only thing that is actually slightly confusing inside what OpenAI does here is that, in addition to this encoder and a decoder they also have something called a **byte encoder** and a **byte decoder**. Unfortunately, this is just a spurious implementation detail and isn't actually deep or interesting in any way. So, I'm going to skip the discussion of it. However, what OpenAI does here, for reasons that I don't fully understand, is that not only do they have this tokenizer, which can encode and decode, but they also have a whole separate layer here in addition that is used serially with the tokenizer. So, you first do **byte encode**, then **encode**, and then you do **decode**, and finally **byte decode**. That's the loop, and they are just stacked serially on top of each other. It's not that interesting, so I won't cover it. You can step through it if you'd like. Otherwise, if you ignore the **byte encoder** and the **byte decoder**, the file will be algorithmically very familiar to you. The meat of it is the **BPE function**, which you should recognize.

```
while True:
    bigram = min(pairs, key = lambda pair: self.bpe_ranks.get(pair, float('inf')))
    if bigram not in self.bpe_ranks:
        break
    first, second = bigram
    new_word = []
    i = 0
    while i < len(word):
        try:
            j = word.index(first, i)
            new_word.extend(word[i:j])
            i = j
        except:
            new_word.extend(word[i:])
            break

        if word[i] == first and i < len(word)-1 and word[i+1] == second:
            new_word.append(first+second)
            i += 2
        else:
            new_word.append(word[i])
            i += 1
    new_word = tuple(new_word)
    word = new_word
    if len(word) == 1:
        break
    else:
        pairs = get_pairs(word)
```

This loop here is very similar to our own **Y loop**, where they're trying to identify the byte pair that they should be merging next. And then, just like we had, they have a **for loop** trying to merge this pair. They go over the entire sequence and merge the pair whenever they find it. They keep repeating that until they run out of possible merges in the text. That's the essence of this file, and there's an **encode** and a **decode function**, just like we have implemented. So, long

story short, what I want you to take away at this point is that, unfortunately, it's a little bit of a messy code that they have. But algorithmically it is identical to what we've built up above and what we've built up above if you understand it is algorithmically what is necessary to actually build a BPE to organizer train it and then both encode and decode

Special tokens, tiktoken handling of, GPT-2/GPT-4 differences

The next topic I would like to turn to is that of **special tokens**. So, in addition to tokens that come from raw bytes and the BPE merges, we can insert all kinds of tokens that we use to delimit different parts of the data or introduce a special structure to the token streams.

```
In [210]: # to download these two files:
          #!wget https://openaipublic.blob.core.windows.net/gpt-2/models/1558M/vocab.bpe
          #!wget https://openaipublic.blob.core.windows.net/gpt-2/models/1558M/encoder.json

          import os, json

          with open('encoder.json', 'r') as f:
              encoder = json.load(f) # <--- ~equivalent to our "vocab"

          with open('vocab.bpe', 'r', encoding="utf-8") as f:
              bpe_data = f.read()
              bpe_merges = [tuple(merge_str.split()) for merge_str in bpe_data.split('\n')[1:-1]]
              # ^----- ~equivalent to our "merges"
```

special tokens

```
In [215]: len(encoder)
```

```
Out[215]: 50257
```

If you look at this **encoder object** from OpenAI's GPT-2, you'll notice that it is very similar to our vocab. The length of this encoder is **50257**. As I mentioned, it's mapping and inverted from the mapping of our vocab. Our vocab goes from integer to string, while they go the other way around for no apparent reason. But the thing to note here is that the mapping table has **50257** entries. Where does that number come from? Well, as I mentioned, there are **256 raw byte tokens**, and then OpenAI actually did **50,000 merges**, which become the other tokens. But this would have been **50256**, so what is the **50257th token**?

```
encoder['<|endoftext|>']
50256
```

There is basically one special token, and that token is called **end-of-text**. It's the very last token. This token is used to delimit documents in the training set. When we create the training data, we have all these documents, and we tokenize them to get a stream of tokens. Those tokens only range from **0 to 50256**. In between those documents, we put a special **end-of-text** token. We use this as a signal to the language model that the document has ended, and what follows is unrelated to the document previously. That said, the language model has to learn this from the data. It needs to learn that this token usually means that it should wipe its sort of memory of what came before. What came before this token is not actually informative for what comes

next, but we are expecting the language model to learn this. We're giving it the special sort of delimiter for these documents.

Tiktokenizer

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Hello world how are you <|endoftext|>

gp12

Token count
116

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Hello world how are you <|endoftext|>

[1640, 1312, 287, 2837, 7, 16, 11, 8949, 2599, 19, 20, 220, 220, 611, 1312, 4064, 513, 6624, 657, 29, 312, 4064, 642, 6624, 657, 25, 198, 220, 220, 220, 0, 220, 220, 220, 3601, 7203, 37, 6457, 48230, 4, 198, 220, 220, 220, 1288, 361, 1312, 4064, 513, 6, 657, 25, 198, 220, 220, 220, 220, 220, 220, 220, 220, 7203, 37, 6457, 4943, 198, 220, 220, 220, 1288, , 1312, 4064, 642, 6624, 657, 25, 198, 220, 220, 220, 220, 220, 220, 3601, 7203, 48230, 4943, 198, , 220, 220, 2073, 25, 198, 220, 220, 220, 220, 220, 20, 220, 3601, 7, 72, 8, 198, 198, 15496, 995, 7, 89, 345, 220, 50256]

We can go here to the tiktokenizer. This is the GPT-2 tokenizer, the code that we've been playing with before. So, we can add the phrase: 'Hello world, world. How are you?' And we're getting different tokens. But now, let's see what happens if I put 'end of text.' You'll notice that until I finish typing, these are all different tokens. When I finally finish it, we get token **50256**. The reason this works is because this didn't actually go through the BPE merges. Instead, the code that actually outputs tokens has special case instructions for handling these special tokens. Unfortunately, we didn't see these special instructions for handling special tokens in the encoder.py. It's absent there. However, if you go to the **tiktoken Library**, which is implemented in Rust, you'll find all kinds of special case handling for these tokens. You can register, create, and add them to the vocabulary. Then, the tokenizer looks for them. Whenever it encounters these special tokens, like 'end of text,' it will actually swap in that special token. These things are outside of the typical BPE encoding algorithm. These special tokens are used pervasively, not just in basic language modeling for predicting the next token in the sequence, but especially when it gets to the fine-tuning stage and all the chatGPT aspects. We don't just want to delimit documents; we want to delimit entire conversations between an assistant and a user. So, if I refresh this tiktokenizer page the default example they have here uses not just base model encoders, but fine-tuned model tokenizers.

Tiktokenizer

System

You are a helpful assistant

X

User

Content

X

Add message

<|im_start|>system

You are a helpful assistant<|im_end|>

<|im_start|>user

<|im_end|>

<|im_start|>assistant

Token count

18

Price per prompt

\$0.000018

<|im_start|>system

You are a helpful assistant<|im_end|>

<|im_start|>user

<|im_end|>

<|im_start|>assistant

[100264, 9125, 198, 2675, 527, 264, 11190, 18328, 100265, 198, 100264, 882, 198, 100265, 198, 100264, 78191, 198]

For example, using the GPT-3.5 Turbo scheme, these are all special tokens: ‘im_start,’ ‘im_end,’ etc. ‘i’ is short for ‘Imaginary,’ and ‘m’ is short for “monologue” by the way. You can see that there’s a sort of start and end for every single message. There can be many other tokens used to delimit these conversations and keep track of the flow of the messages.

Extending tiktoken

You may wish to extend `tiktoken` to support new encodings. There are two ways to do this.

Create your `Encoding` object exactly the way you want and simply pass it around.

```
cl100k_base = tiktoken.get_encoding("cl100k_base")

# In production, load the arguments directly instead of accessing private attributes
# See openai_public.py for examples of arguments for specific encodings
enc = tiktoken.Encoding(
    # If you're changing the set of special tokens, make sure to use a different name
    # It should be clear from the name what behaviour to expect.
    name="cl100k_im",
    pat_str=cl100k_base._pat_str,
    mergeable_ranks=cl100k_base._mergeable_ranks,
    special_tokens={
        **cl100k_base._special_tokens,
        "<|im_start|>": 100264,
        "<|im_end|>": 100265,
    }
)
```

Now, let’s go back to the Tik Token library. When you scroll to the bottom, they discuss how you can extend Tik Token. You can fork the CL 100K base tokenizers in GPT-4. For example, you can extend it by adding more special tokens. These are totally up to you; you can come up with any arbitrary tokens and add them with a new ID. The Tik Token library will correctly swap them out when it encounters them in the strings.


```
def gpt2():
    mergeable_ranks = data_gym_to_mergeable_bpe_ranks(
        vocab_bpe_file="https://openaipublic.blob.core.windows.net/gpt-2/encodings/main/vocab.bpe",
        encoder_json_file="https://openaipublic.blob.core.windows.net/gpt-2/encodings/main/encoder",
        vocab_bpe_hash="1ce1664773c50f3e0cc8842619a93edc4624525b728b188a9e0be33b7726adc5",
        encoder_json_hash="196139668be63f3b5d6574427317ae82f612a97c5d1cdaf36ed2256dbf636783",
    )
    return {
        "name": "gpt2",
        "explicit_n_vocab": 50257,
        # The pattern in the original GPT-2 release is:
        # r''''s|'t|'re|'ve|m|'ll|'d| ?[\p{L}]+| ?[\p{N}]]+| ?[\s\p{L}\p{N}]+\s+(?!S)|\s+'''
        # This is equivalent, but executes faster:
        "pat_str": r''''(?:[sdmt]|ll|ve|re)| ?[\p{L}]+| ?[\p{N}]]+| ?[\s\p{L}\p{N}]+\s+(?!S)|\s+'''",
        "mergeable_ranks": mergeable_ranks,
        "special_tokens": {ENDOFTEXT: 50256},
    }
```

Additionally, let's revisit the file we looked at previously. In GPT-2 and Tik Token OpenAI, we have the vocabulary and the pattern for splitting. Here, **"special_tokens"**: we register the single special token in GPT-2, which was the 'end of text' token. We saw that it has this ID. In GPT-4, when they define this, you'll notice that the pattern has changed, as we've discussed. Also, the special tokens have changed in this tokenizer. Of course, we still have the 'end of text' token, just like in GPT-2, but we also see three (or four) additional tokens here.

```
def cl100k_base():
    mergeable_ranks = load_tiktoken_bpe(
        "https://openaipublic.blob.core.windows.net/encodings/cl100k_base.tiktoken",
        expected_hash="223921b76ee99bde995b7ff738513eef100fb51d18c93597a113bcffe865b2a7",
    )
    special_tokens = {
        ENDOFTEXT: 100257,
        FIM_PREFIX: 100258,
        FIM_MIDDLE: 100259,
        FIM_SUFFIX: 100260,
        ENDOFPROMPT: 100276,
    }
    return {
        "name": "cl100k_base",
        "pat_str": r''''(?:[sdmt]|ll|ve|re)|[^\r\n\p{L}\p{N}]?*\p{L}+|\p{N}{1,3}| ?[\s\p{L}\p{N}]]+| ?[\s\p{L}\p{N}]+\s+(?!S)|\s+'''",
        "mergeable_ranks": mergeable_ranks,
        "special_tokens": special_tokens,
    }
```

Here, the **Fim_prefix, middle, and suffix**—what is **fim**? **Fim** is short for 'fill in the middle.' If you'd like to learn more about this idea, it comes from this paper (um, and I'm not going to go into detail in this video; it's beyond this video). Then there's one additional **special token** **"ENDOFTEXT"**. So, that's the encoding as well. It's very common, basically, to train a language model. And if you'd like, you can add special tokens. Now, when you add special tokens, you, of course, have to do some model surgery to the Transformer and all the parameters involved in that Transformer. Why? Because you're basically adding an integer, and you want to make sure that, for example, your embedding matrix for the vocabulary tokens has to be extended by adding a row. Typically, this row would be initialized with small random numbers or something like that because we need to have a vector that now stands for that token. In addition to that, you have to go to the final layer of the Transformer and ensure that the projection at the very end into the classifier is extended by one as well. So, basically, there's some model surgery involved that you have to couple with the tokenization changes if you are going to add special

tokens. But this is a very common operation that people do, especially if they'd like to fine-tune the model. For example, taking it from a base model to a chat model like Chat GPT-4. So, at this point, you should have everything you need in order to build your own GPT-4 tokenizer.

minbpe exercise time! write your own GPT-4 tokenizer

Now, in the process of developing this lecture, I've done that and I published the code under this repository. MBP looks like this right now as I'm recording, but the MBP repository will probably change quite a bit because I intend to continue working on it. In addition to the MBP repository, I've published this exercise progression that you can follow. If you go to exercise.md, you'll find me breaking up the task ahead of you into four steps that build up to what can be a GPT-4 tokenizer. Feel free to follow these steps exactly and refer to the MBP repository whenever you feel stuck. The tests could be useful, or you can explore the MBP repository itself. I've tried to keep the code fairly clean and understandable, so feel free to reference it whenever you get stuck.

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base") # GPT-4 tokenizer
print(enc.encode("안녕하세요 🌻 (hello in Korean!)"))
print(enc.decode(enc.encode("안녕하세요 🌻 (hello in Korean!)")) == "안녕하세요 🌻 (hello in Korean!)")
# match the above for your own tokenizer, and also implement a train() function

[31495, 230, 75265, 243, 92245, 62904, 233, 320, 15339, 304, 16526, 16715]
True
```

Additionally, once you write it, you should be able to reproduce this behavior using Tiktoken. By getting the GPT-4 tokenizer, you can encode the string and obtain these tokens. You can then encode and decode the exact same string to recover it. Furthermore, you should be able to implement your own train function, which the Tiktoken Library does not provide.

Now, in the process of developing this lecture, I've done that and I published the code under this repository. MBP looks like this right now as I'm recording, but the MBP repository will probably change quite a bit because I intend to continue working on it. In addition to the MBP repository, I've published this exercise progression that you can follow. If you go to exercise.md, you'll find me breaking up the task ahead of you into four steps that build up to what can be a GPT-4 tokenizer. Feel free to follow these steps exactly and refer to the MBP repository whenever you feel stuck. The tests could be useful, or you can explore the MBP repository itself. I've tried to keep the code fairly clean and understandable, so feel free to reference it whenever you get stuck. Additionally, once you write it, you should be able to reproduce this behavior using Tiktoken. By getting the GPT-4 tokenizer, you can encode the string and obtain these tokens. You can then encode and decode the exact same string to recover it. Furthermore, you should be able to implement your own train function, which the Tiktoken Library does not provide.

sentencepiece library intro, used to train Llama 2 vocabulary

Okay, so we are now going to move on from tiktoken and the way that OpenAI tokenizes its strings. We're going to discuss one more very commonly used library for working with tokenization in language models.

And that is **SentencePiece**. SentencePiece is very commonly used in language models because, unlike TikToken, it can do both training and inference, and is quite efficient at both. It supports a number of algorithms for training vocabularies, but one of them is the **BPE (Byte Pair Encoding)** algorithm that we've been looking at. So, SentencePiece supports it.

Now, SentencePiece is used both by **LAMA** and **MISTRAL series**, as well as many other models. It is on GitHub under **Google/ SentencePiece**. The big difference with SentencePiece, and we're going to look at an example because this is kind of hard and subtle to explain, is that they think differently about the order of operations here.

In the case of TikToken, we first take our code points in the string, encode them using **UTF-8**, and then we're merging bytes. It's fairly straightforward. For SentencePiece, it works directly on the level of the code points themselves. So, it looks at whatever code points are available in your training set, and then it starts merging those code points. The BPE is running on the level of code points. And if you happen to run out of code points (there are maybe some rare code points that just don't come up too often, and the rarity is determined by this **character coverage** hyperparameter), then these code points will either get mapped to a special unknown token like **UNK**, or if you have the **byte fallback** option turned on, then that will take those rare code points and encode them using UTF-8.

And then the individual bytes of that encoding will be translated into tokens, and there are these special **byte tokens** that basically get added to the vocabulary. So, it uses **BPE (Byte Pair Encoding)** on the code points, and then it falls back to bytes for rare code points. That's kind of the difference.

Personally, I find the TikToken way significantly cleaner, but it's a subtle but pretty major difference between the way they approach tokenization. Let's work with a concrete example because otherwise, this is kind of hard to get your head around.

```
# write a toy.txt file with some random text
with open("toy.txt", "w", encoding="utf-8") as f:
    f.write("SentencePiece is an unsupervised text tokenizer and detokenizer mainly for Neural Network-based text gene
```

So, let's work with a concrete example. This is how we can import **SentencePiece**, and then here we're going to take (I think I took) the description of SentencePiece, and I just created a little toy data set. It really likes to have a file, so I created a toy.txt file with this content.

What's kind of a little bit crazy about SentencePiece is that there are a ton of options and configurations. The reason this is so is because SentencePiece has been around for a while, and it really tries to handle a large diversity of things. Because it's been around, I think it has quite a

bit of accumulated historical baggage as well. In particular, there's like a ton of configuration arguments.

There's also quite useful documentation when you look at the raw **protobuf** that is used to represent the trainer spec and so on. Many of these options are irrelevant to us, so maybe to point out one example: – **shrinking_factor**. This shrinking factor is not used in the **BPE (Byte Pair Encoding)** algorithm, so this is just an argument that is irrelevant to us. It applies to a different training algorithm.

```
options = dict(
    # input spec
    input="toy.txt",
    input_format="text",
    # output spec
    model_prefix="tok400", # output filename prefix
    # algorithm spec
    # BPE alg
    model_type="bpe",
    vocab_size=400,
    # normalization
    normalization_rule_name="identity", # ew, turn off normalization
    remove_extra_whitespaces=False,
    input_sentence_size=200000000, # max number of training sentences
    max_sentence_length=4192, # max number of bytes per sentence
    seed_sentencepiece_size=1000000,
    shuffle_input_sentence=True,
    # rare word treatment
    character_coverage=0.99995,
    byte_fallback=True,
    # merge rules
    split_digits=True,
    split_by_unicode_script=True,
    split_by_whitespace=True,
    split_by_number=True,
    max_sentencepiece_length=16,
    add_dummy_prefix=True,
    allow_whitespace_only_pieces=True,
    # special tokens
    unk_id=0, # the UNK token MUST exist
    bos_id=1, # the others are optional, set to -1 to turn off
```

Now, what I tried to do here is set up SentencePiece in a way that is very, very similar (as far as I can tell, maybe identical, hopefully) to the way that **LAMA 2** was trained. The way they trained their own tokenizer. I did this by taking the tokenizer model file that Meta released, opening it using the **protobuf** file that you can generate, and then inspecting all the options. I tried to copy over all the options that looked relevant. So, where it says “**input=“toy.txt”**” we set up the input (it's raw text in this file), and the output is going to be **prototok400.model and vocab**. We're saying that we're going to use the BPE algorithm, and we want a **BPE size of 400**. Then there's a ton of configurations here for basically pre-processing and normalization rules, as they're called. Normalization used to be very prevalent, I would say, before language models in natural language processing.

So, in machine translation, text classification, and so on, you want to normalize and simplify the text. You want to turn it all lowercase and remove all double whitespaces, etc. However, in language models, we prefer not to do any of it, or at least that is my preference as a deep learning person. You want to not touch your data and keep the raw data as much as possible. In a raw form, you're basically trying to turn off a lot of this if you can.

The other thing that **SentencePiece** does is that it has this concept of sentences. SentencePiece was developed, I think, early in the days when there was an idea that you're training a tokenizer on a bunch of independent sentences. So, it has a lot of options related to how many sentences you're going to train on, what is the maximum sentence length, shuffling sentences, and so forth. For SentencePiece, sentences are kind of like the individual training examples. However, in the context of language models, I find that this is a very sparse and weird distinction. Sentences are just like 'don't touch the raw data.' Sentences happen to exist, but in raw datasets, there are a lot of ambiguities like what exactly is a sentence and what isn't a sentence. and so, I think it's really hard to define what an actual sentence is if you really dig into it. There could be different concepts of it in different languages or something like that. So, why even introduce the concept? It honestly doesn't make sense to me. I would just prefer to treat a file as a giant stream of bytes.

SentencePiece has a lot of treatment around rare word characters (and when I say 'word,' I mean code points). We'll come back to this in a second. It also has a lot of other rules for basically splitting digits, splitting whitespace and numbers, and how you deal with that. These are some kind of merge rules. I think this is a little bit equivalent to tiktoken using regular expressions to split up categories. There's a kind of equivalence of it if you squint at it in SentencePiece. For example, you can also split up the digits and so on.

There are a few more things here that I'll come back to in a bit. And then there are some special tokens that you can indicate. It hardcodes the UNK token, the beginning of sentence, end of sentence, and a pad token. The UNK token must exist, from my understanding. And then some other things.

We can train, and when I press 'train,' it's going to create this file: tok400.model and tok400.vocab. I can then load the model file and inspect the vocabulary. We trained with a vocab size of 400 on this text here, and these are the individual pieces, the individual tokens that SentencePiece will create: In the beginning, we see that we have the **UNK token** with the ID zero. Then we have the **beginning of sequence (BOS)** and **end of sequence (EOS)** tokens with IDs one and two, respectively. We set the **pad ID** to negative one, so there's no pad ID here.

```
[['<unk>', 0],  
 ['<s>', 1],  
 ['</s>', 2],  
 ['<0x00>', 3],  
 ['<0x01>', 4],  
 ['<0x02>', 5],  
 ['<0x03>', 6],  
 ['<0x04>', 7],  
 ...]
```

Next, we have the individual **byte tokens**. Here, we saw that **byte_fallback** in LLM was turned on (so it's true). What follows are going to be the 256 byte tokens, and these are their IDs.

```

['er', 259],
['_t', 260],
['ce', 261],
['in', 262],
['ra', 263],
['_a', 264],
['de', 265],
['er', 266],
['_s', 267],
['ent', 268],
['or', 269],

```

After the byte tokens, we see the **merges**. These are the parent nodes in the merges. We're not seeing the children; we're just seeing the parents and their IDs.

```

['oc', 360],
['e', 361],
['_', 362],
['n', 363],
['t', 364],
['i', 365],
['r', 366],
['a', 367],
['o', 368],
['s', 369],
['d', 370],
['c', 371],
['l', 372],
['u', 373],
['g', 374],
['m', 375],

```

Finally, after the merges, we have the individual **tokens** and their IDs. These are the individual **code point tokens**, if you will. They come at the end. So, that is the ordering with which SentencePiece sort of represents its vocabularies: **special tokens**, **byte tokens**, **merge tokens**, and **individual code point tokens**. All these raw code points to tokens are the ones that it encountered in the training set, so those individual code points are the entire set of code points that occurred here.

And then those that are extremely rare, as determined by character coverage. So, if a code point occurred only a single time out of, like, a million um sentences or something like that, then it would be ignored, and it would not be added to our vocabulary. Once we have a vocabulary, we can encode it into IDs, and we can sort of get a list. And then here, I am also decoding the individual tokens back into little pieces, as they call it.

```

ids = sp.encode("hello안녕하세요")
print(ids)
[362, 378, 361, 372, 358, 362, 239, 152, 139, 238, 136, 152, 240, 152, 155, 239, 135, 187, 239, 157, 151]

```

So let's take a look at what happened here: Hello, space annyeonghaseyo. These are the token IDs we got back. When we look here, a few things sort of jump to mind. Number one, take a look at these characters. The Korean characters, of course, were not part of the training set. So SentencePiece is encountering code points that it has not seen during training time, and those code points do not have a token associated with them. So suddenly, these are unknown tokens. But because **byte_fallback** is set to true, instead, SentencePiece falls back to bytes. It encodes it

with UTF-8, and then it uses these tokens to represent those bytes. That's what we are getting sort of here.

```
[362, 378, 361, 372, 358, 362, 239, 152, 139, 238, 136, 152, 240, 152, 155, 239, 135, 187,
```

This is the UTF-8 encoding, and in this shifted by three, uh, because of these special tokens here that have IDs earlier on. So that's what happened here. Now, one more thing: before I go on with respect to the `byte_fallback`, if we remove `byte_fallback`. If this is false, what's going to happen? Let's retrain. The first thing that happened is all the byte tokens disappeared, right?

And now we just have the merges, and we have a lot more merges now because we have a lot more space. We're not taking up space in the vocab size with all the bytes. Now, if we encode this, we get a zero. So this entire string here suddenly has no `byte_fallback`, so this is unknown. And unknown is **unk**, so this is zero because the **unk** token is token zero. You have to keep in mind that this would feed into your language model.

So what is a language model supposed to do when all kinds of different things that are unrecognized, because they're rare, just end up mapping into Unk? It's not exactly the property that you want. That's why I think Llama correctly used `byte_fallback` as true because we definitely want to feed these unknown or rare code points into the model in some manner.

```
ids = sp.encode("hello 안녕하세요")
print(ids)

[362, 378, 361, 372, 358, 362, 239, 152, 139, 238, 136, 152, 240, 152, 155, 239, 135, 187, 0]

print([sp.id_to_piece(idx) for idx in ids])

['_ ', 'h', 'e', 'l', 'l', 'o', ' ', '<0xEC>', '<0x95>', '<0x88>', '<0xEB>', '<0x85>', '<0x98>', '<0xEC>', '<0x84>', '<0xB8>', '<0xEC>', '<0x9A>', '<0x94>']
```

The next thing I want to show you is the following notice: When we are decoding all the individual tokens, you see how spaces between “**hello**” and “안녕하세요” ends up being this bold underline. I'm not 100% sure, by the way, why SentencePiece switches whitespace into these bold underscore characters. Maybe it's for visualization. I'm not 100% sure why that happens. But notice this: Why do we have an extra space in the front of 'hello'? where is this coming from? Well, it's coming from this option: **add_dummy_prefix** is true. When you go to the documentation, it says, 'Adds dummy whitespaces at the beginning of text in order to treat “World” in “world” and “hello world” in the exact same way.' So what this is trying to do is the following: If we go back to our tiktokenizer, 'world' as a token by itself has a different ID than space world.

```
world
hello world
```

```
[14957, 198, 15339, 1917]
```

So we have this is 14957, but the other space is 1917, etc. So these are two different tokens for the language model. The language model has to learn from data that they are actually kind of like a very similar concept. So, to the language model, the Tiktoken world—basically words in the beginning of sentences—and words in the middle of sentences actually look completely different. It has to learn that they are roughly the same.

So, this **'add_dummy_prefix'** is trying to fight that a little bit. The way that works is that it basically adds a dummy prefix. As part of pre-processing, it will take the string and add a space. This is done in an effort to make **'world'** and **'hello world'** the same—they will both be 'space world.' So, that's one other kind of pre-processing option that is turned on. Llama 2 also uses this option, and that's everything that I want to say for my preview of SentencePiece and how it is different.

My summary for SentencePiece from all of this is: number one, I think that there's a lot of historical baggage in SentencePiece, a lot of concepts that I think are slightly confusing. I think potentially, um, they contain foot guns, like this concept of a sentence and its maximum length and stuff like that. Otherwise, it is fairly commonly used in the industry because it is efficient and can do both training and inference. It has a few quirks, like, for example, unk token must exist and the way the byte fallbacks are done and so on. I don't find it particularly elegant, and unfortunately, I have to say it's not very well documented. So, it took me a lot of time working with this myself and just visualizing things and trying to really understand what is happening here because the documentation, unfortunately, is in my opinion not super amazing. But it is a very nice repo that is available to you if you'd like to train your own tokenizer right now.

[how to set vocabulary set? revisiting gpt.py transformer](#)

Okay, let me now switch gears again as we're starting to slowly wrap up here. I want to revisit this issue in a bit more detail of how we should set the vocab size and what are some of the considerations around it. So for this, I'd like to go back to the model architecture that we developed in the last video when we built the GPT from scratch. This here was the file that we constructed in the previous video, and we defined the Transformer model.

Let's specifically look at vocab size and where it appears in this file.


```

chars = sorted(list(set(text)))
vocab_size = len(chars)
# create a mapping from characters to integers

```

Here, we define the vocab size; at this time, it was 65 or something like that—an extremely small number. However, this will grow much larger. You’ll notice that vocab size doesn’t come up too frequently in most of these layers.

```

def __init__(self):
    super().__init__()
    # each token directly reads off the logits for the next token from a lookup table
    self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
    self.position_embedding_table = nn.Embedding(block_size, n_embd)
    self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
    self.ln_f = nn.LayerNorm(n_embd) # final layer norm
    self.lm_head = nn.Linear(n_embd, vocab_size)

```

The only places it surfaces are exactly these two spots. First, when we define the language model, there’s the token embedding table—a two-dimensional array where the vocab size essentially determines the number of rows. Each vocabulary element (each token) has a vector that we’ll train using **backpropagation**. This vector is of size ‘embed,’ which corresponds to the number of channels in the Transformer. As the vocab size increases, this embedding table, as I mentioned earlier, will also expand. We’ll be adding rows to it.

```

def __init__(self):
    super().__init__()
    # each token directly reads off the logits for the next token from a lookup table
    self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
    self.position_embedding_table = nn.Embedding(block_size, n_embd)
    self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
    self.ln_f = nn.LayerNorm(n_embd) # final layer norm
    self.lm_head = nn.Linear(n_embd, vocab_size)

```

```

# idx and targets are both (B,T) tensor of integers
tok_emb = self.token_embedding_table(idx) # (B,T,C)
pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
x = tok_emb + pos_emb # (B,T,C)
x = self.blocks(x) # (B,T,C)
x = self.ln_f(x) # (B,T,C)
logits = self.lm_head(x) # (B,T,vocab_size)

```

At the end of the Transformer, there’s the ‘LM head’ layer, which is a linear layer. You’ll notice that this layer is used at the very end to produce the logits—these logits become the probabilities for the next token in the sequence.

So, intuitively, we’re trying to produce a probability for every single token that might come next at every point in time of that Transformer. If we have more and more tokens, we need to produce more and more probabilities. So, every single token is going to introduce an additional dot product that we have to compute here in this linear layer for the final layer in a Transformer. Now, why can’t vocab size be infinite? Why can’t we grow to infinity?

Well, number one: Your token embedding table is going to grow, and your linear layer is going to expand. Consequently, we'll be performing a lot more computation here because this 'LM head' layer will become more computationally expensive.

Number two: Having more parameters might lead to concerns about undertraining some of these parameters. Intuitively, if you have a very large vocabulary size—for instance, a million tokens—then each of these tokens will appear more and more rarely in the training data. There are many other tokens scattered throughout, so we'll encounter fewer and fewer examples for each individual token. As a result, you might worry that the vectors associated with every token will be undertrained because they don't participate frequently in the forward-backward pass.

Additionally, as your vocab size grows, your sequences will shrink significantly. While this allows us to attend to more and more text, there's also the concern that overly large chunks are being compressed into single tokens. Consequently, the model may not have sufficient time to think through each segment of characters in the text. You can think about it in that way, right?

So basically, we're squishing too much information into a single token. Then, the forward pass of the Transformer is not enough to actually process that information appropriately. These are some of the considerations you're thinking about when you're designing the vocab size. As I mentioned, this is mostly an empirical hyperparameter. It seems that in state-of-the-art architectures today, this vocab size is usually in the high 10,000s or somewhere around 100,000.

The next consideration I want to briefly talk about is: What if we want to take a pre-trained model and extend the vocab size? This is done fairly commonly, actually. For example, during fine-tuning for ChatGPT, many new special tokens get introduced on top of the base model. These special tokens help maintain the metadata and structure of conversation objects between a user and an assistant. So, it takes quite a few special tokens. You might also try to throw in more special tokens—for instance, for using the browser or any other tool. It's very tempting to add tokens for all kinds of special functionality.

If you want to add a token, that's totally possible. All we have to do is resize this embedding. We add rows and initialize these parameters from scratch with small random numbers. Then, we extend the weight inside this linear layer. We start making dot products with the associated parameters to calculate the probabilities for these new tokens. Both of these are just resizing operations. It's a very mild model surgery and can be done fairly easily. It's quite common that, basically, you would freeze the base model, introduce these new parameters, and then only train these new parameters to incorporate new tokens into the architecture. You can freeze arbitrary parts of it or train arbitrary parts of it—that's totally up to you. But basically, minor surgery is required if you'd like to introduce new tokens.

training new tokens, example of prompt compression

Finally, I'd like to mention that there's actually an entire design space of applications in terms of introducing new tokens into a vocabulary. This goes way beyond just adding special tokens and

new functionality. Let me give you a sense of this design space, although it could easily be an entire video on its own.

[This](#) paper discusses learning to compress prompts using what they call ‘gist tokens.’ The rough idea is as follows: Imagine you’re using language models in a setting that requires very long prompts. These lengthy prompts slow down everything because you have to encode them, use them, and then attend to them. It becomes cumbersome to deal with such large prompts.

Instead, in this paper, they introduce new tokens. Think of it as having a few fresh tokens that you insert into a sequence. Then, you train the model through distillation. Essentially, you freeze the entire model and only train the representations of these new tokens—their embeddings. You optimize these new tokens so that the behavior of the language model becomes identical to the model that originally had a very long prompt tailored for your needs.

This approach serves as a compression technique, condensing that lengthy prompt into just a handful of new ‘gist’ tokens. During testing, you can discard your old prompt and simply swap in these tokens. Remarkably, they stand in for the extensive prompt and yield almost identical performance.

So, this is one technique—a class of parameter-efficient fine-tuning techniques—where most of the model is basically fixed. There’s no training of the model weights, no training of LORA, or anything like that involving new parameters. The parameters you’re now training are just the token embeddings.

The next thing I want to briefly address is that, recently, there’s a lot of momentum in how you could construct Transformers that can simultaneously process not just text as the input modality, but also a variety of other modalities. Be it images, videos, audio, etc., the question is: How do you feed in all these modalities and potentially predict these modalities from a Transformer? Do you have to change the architecture in some fundamental way?

I think what a lot of people are starting to converge towards is that you’re not changing the architecture; you stick with the Transformer. You just kind of tokenize your input domains and then call it a day, pretending it’s just text tokens, and do everything else in an identical manner.

So here, for example, there was an early paper that has a nice graphic for how you can take an image and chunk it into integers. These will basically become the tokens of images, as an example. These tokens can be hard tokens, where you force them to be integers, or they can also be soft tokens, where you don’t require these to be discrete but do force these representations to go through bottlenecks, like in autoencoders.

Also, in this paper that came out from OpenAI, **SORA**, which I think really blew the minds of many people and inspired a lot of people in terms of what’s possible, they have a graphic here. They talk briefly about how **LLMs** (Language and Vision Models) have text tokens. **SORA** has visual patches. So, again, they came up with a way to chunk videos into basically tokens when they own vocabularies. You can either process discrete tokens, say with autoregressive models,

or even soft tokens with diffusion models. All of that is sort of being actively worked on, designed, and is beyond the scope of this video. But it's something I wanted to mention briefly.

revisiting and explaining the quirks of LLM tokenization

Okay, now that we have delved quite deep into the tokenization algorithm and gained a better understanding of how it works, let's loop back to the beginning of this video. We'll go through some of these bullet points and explore why they occur.

Firstly, why can't my language model (LLM) spell words very well or perform other spelling-related tasks? Fundamentally, this issue arises because the characters are chunked up into tokens, and some of these tokens are actually quite long. As an example, I examined the GP4 vocabulary and focused on one of the longer tokens: 'default style.' It turns out that 'default style' is treated as a single individual token. That's a lot of characters for just one token.

My suspicion is that there's simply too much information crammed into this single token, which affects the model's ability to handle spelling tasks related to it. To illustrate this, I asked the model how many letters 'l' are there in the word '**.DefaultCellStyle**'. Keep in mind that the prompt intentionally framed 'default style' as a single token. The model's response indicated that it believes there are three 'l's, but in reality, there are four.

So, unfortunately, the model's performance in this regard didn't go extremely well. Let's look at another kind of character-level task. For example, here I asked GP4 to reverse the string 'default style.' It attempted to use a code interpreter, but I interrupted it and instructed it to proceed anyway. Surprisingly, it produced gibberish—indicating that it doesn't actually know how to reverse this string from right to left. The result was incorrect.

Continuing with my working hypothesis that this issue might be related to tokenization, I tried a different approach. I said, 'Okay, let's reverse the exact same string, but follow this approach: Step one, print out every single character separated by spaces. Then, as step two, reverse that list.' Again, GP4 attempted to use a tool, but when I halted it, it correctly produced all the characters in order. Subsequently, it reversed them accurately.

Interestingly, it seems that GP4 struggles to reverse the string directly. However, when we break it down by listing out individual characters first, it becomes much easier for the model to recognize these individual tokens, reverse them, and print them out. Quite intriguing, isn't it?

So let's continue. Now, why are LLMs worse at non-English languages? I briefly covered this already, but basically, it's not only that the language model sees less non-English data during training of the model parameters, but also the tokenizer is not sufficiently trained on non-English data.

For example, the phrase 'hello, how are you' consists of five tokens, while its translation is 15 tokens. This results in a threefold increase. Similarly, 'annyeonghaseyo' which means 'hello' in Korean, ends up being three tokens. Surprisingly, our English 'hello' is a single token. Essentially,

everything becomes more bloated and diffuse, which partly explains why the model performs worse on other languages.



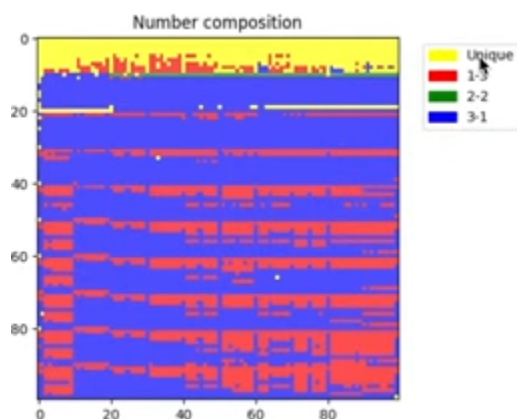
The image shows a handwritten addition problem: 1,296 plus 3,457 equals 53. The digits '1' and '1' in the numbers are highlighted in purple and green respectively. To the right of the problem, three steps are listed: 'Step 1. Add the ones', 'Step 2. Add the tens', and 'Step 3. Add the hundreds'.

$$\begin{array}{r} 1,296 \\ +3,457 \\ \hline 53 \end{array}$$

Step 1. Add the ones
Step 2. Add the tens
Step 3. Add the hundreds

Now, let's address why LLMs struggle with simple arithmetic. This issue relates to the tokenization of numbers. Notice that addition follows a character-level algorithm. For instance, when adding numbers, we start with the ones, then the tens, and finally the hundreds. You have to refer to specific parts of these digits. However, these numbers are represented arbitrarily based on whatever happened during the tokenization process—whether they merged or remained separate.

There's an entire blog post about this that I think is quite good. Integer tokenization is insane, and this person systematically explores the tokenization of numbers. I believe this is related to GPT-2.



They notice that, for example, with four-digit numbers, you can examine whether it is a single token or whether it consists of two tokens. Specifically, it could be a 1-3 combination, a 2-2 combination, or even a 3-1 combination. All these different numbers result from various combinations, and you can imagine that this process is entirely arbitrary. The model unfortunately sometimes interprets four tokens for all four digits, other times three, two, or even just one token. It's done in an arbitrary manner, which poses a challenge for the language model. It's incredible that the model can handle this variation, but it's not ideal.

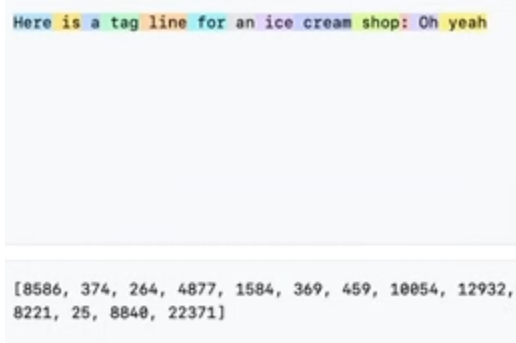
For example, when training the LLM-2 algorithm, Meta uses SentencePiece to split up all the digits. This approach aims to improve simple arithmetic performance. And finally, why is GPT-2 not as good in Python? Again, this is partly a modeling issue, both in the architecture and the dataset, as well as the strength of the model. However, it's also partially due to tokenization. As we saw in the simple Python example, the encoding efficiency of the tokenizer for handling spaces in Python is terrible. Every single space is treated as an individual token, dramatically reducing the context length that the model can attend to. This is almost like a tokenization bug for GPT-2, which was later fixed with GPT-4.

Now, here's another interesting observation: my LLM abruptly halts when it encounters the string '`<|endoftext|>`'. Let me illustrate this behavior. When I instruct it to print '`<|endoftext|>`,' it responds by asking me to specify the string. Even when I explicitly provide '`<|endoftext|>`' as the string, it still fails to print anything. Clearly, there's an issue related to the handling of this special token.

And I don't actually know what OpenAI is doing under the hood. Here, whether they are potentially parsing this as an actual token instead of just being the end of text. sort of like as individual pieces of it without the special token handling logic. So, it might be that when someone is calling `.encode`, they are passing in the allowed special tokens. They are allowing "`<|endoftext|>`" as a special character in the user prompt. However, the user prompt, of course, is attacker-controlled text. You would hope that they don't really parse or use special tokens from that kind of input. But it appears that something is definitely going wrong here. Your knowledge of these special tokens ends up being a potential attack surface. If you'd like to confuse language models, try giving them some special tokens and see if you're breaking something by chance.

Now, let's discuss the trailing whitespace issue. If you visit the Playground and go to GPT-3.5 Turbo instruct (which is not a chat model but a completion model), think of it as being closer to a base model. It does completion and continues the token sequence. Here's a tagline for an ice cream shop, and we want to continue the sequence. When we submit, we get a bunch of tokens—no problem. But now, suppose I do this: instead of pressing submit, I add a space after "ice cream shop" and then click submit.

We get a warning: your text ends in a trailing space, which causes worse performance due to how the API splits text into tokens. So, what's happening here? It still gave us a sort of completion, but let's take a look at what's happening.



The screenshot shows the OpenAI Playground interface. The input field contains the text "Here is a tag line for an ice cream shop: Oh yeah" with a trailing space. Below the input field, the output shows the tokenized sequence: [8586, 374, 264, 4877, 1584, 369, 459, 10054, 12932, 8221, 25, 8840, 22371].

Here's a tagline for an ice cream shop. Now, what does this look like in the actual training data? Suppose you found the completion in a training document somewhere on the internet, and the language model (LLM) was trained on this data. Maybe it's something like, 'Oh yeah'—though admittedly, it's a terrible tagline. But notice that when I create 'o,' you see that because there's a space character, it's always a prefix to these tokens in GPT. So, it's not an 'O' token; it's a 'space o' token. The space is part of the 'O,' and together, they form token 8840 (that's 'space o'). What's happening here is that when I just have it without the "Oh yeah" and let it complete the next token, it can sample the 'space o' token. However, if I add my space, then when I encode this string, I have, 'Here's a tagline for an ice cream shop,' and this space at the very end becomes a token (token 220).

```
Here is a tag line for an ice cream shop:
[8586, 374, 264, 4877, 1584, 369, 459, 10054, 12932,
8221, 25, 220]
```

And so we've added token 220. This token would otherwise be part of the tagline. Because if there actually is a tagline here, 'space o' is the token. This sudden distribution shift occurs because this space is part of the next token. However, we're placing it here like this, and the model has seen very little data of actual 'Space' by itself. We're asking it to complete the sequence by adding more tokens, but the problem is that we've sort of begun the first token, and now it's been split up. Consequently, we're out of the expected distribution, leading to arbitrary issues. This situation is quite rare for the model to encounter, which is why we receive the warning. The fundamental issue here is that the language model (LLM) operates on these tokens, which are text chunks—not characters in the way you and I would typically think of them. These tokens serve as the building blocks of what the LLM processes, resulting in some peculiar outcomes. Let's revert to our default cell style. I'd wager that the model has never encountered 'default cell sta' without 'Le' in there.

```
.DefaultCellSty
[13578, 3683, 626, 88]
```

It's always seen this as a single group because this is some kind of a function. Um, I'm guessing I don't actually know what this is part of; it seems to be related to an API. However, I bet you that

it's never encountered this combination of tokens in its training data. Or, I think it would be extremely rare. So, I took this and copy-pasted it to GPT playground. Then I tried to complete from it, but immediately, it gave me a big error. it said "The model predicted a completion that begins with a stop sequence, resulting in no output. Consider adjusting your prompt or stop sequences." What happened here when I clicked submit is that the model emitted something like an 'end of text' token, I believe. Essentially, it predicted the stop sequence right away, leaving no room for further completion. Hence, I'm receiving a warning again because we're off the expected data distribution. The model is just predicting arbitrary things—it's quite confused. This situation is akin to giving it brain damage; it's never seen this before, and it's shocked. I tried it again with different stuff mixed with it, and in this case, it completed it.

```
.DefaultCellStyle.PaddingBottom = 2f;  
cellTitle = new PdfPCell(new Phrase("", font));
```



However, for some reason, this request may violate our usage policies; it was flagged. Basically, something went wrong, and there's some 'jank.' You can just feel the 'jank' because the model is extremely unhappy with just this input, and it doesn't know how to complete it since it's never occurred in the training set.

In a training set, it always appears as ".DefaultCellStyle" and becomes a single token. These kinds of issues arise when tokens are either completing the first character of the next token or when long tokens have some characters cut off. All of these are partial token issues, in my description. If you delve into the Tiktoken repository and explore the Rust code, search for 'unstable,' and you'll encounter code related to 'unstable tokens.' Curiously, none of this information about unstable tokens is documented anywhere, yet there's a substantial amount of code handling them. Unstable tokens align precisely with what I'm describing here. Ideally, for a completion API, we'd want something more sophisticated. For instance, when we input

'defaultcellsta' and request the next token sequence, we're not necessarily trying to append the next token directly after this list.

We're actually trying to append—we're trying to consider lots of tokens. If we were, or I guess, if we're trying to search over characters that, if retained, would be of high probability. If that makes sense, we can actually add a single individual character instead of just adding the next full token that comes after this partial token list. So, this is very tricky to describe, and I invite you to maybe look through this. It ends up being extremely gnarly and hairy, kind of a topic. It comes from tokenization fundamentally. Maybe I can even spend an entire video talking about unstable tokens sometime in the future. Okay, and I'm really saving the best for last: my favorite one by far is the solid gold Magikarp. It just—okay, so this comes from this blog post, 'Solid Gold Magikarp.' It's internet-famous now for those of us in LLMs and Basically, I would advise you to read this blog post in full. But let me summarize what this person was doing: they went to the token embedding stable and clustered the tokens based on their embedding representation. This person noticed that there's a cluster of tokens that look really strange.



'22'	'ortunately'	'getting'	'ing'	'cells'	'attRot'
'26'	'However'	'creating'	'es'	'models'	'e'
'38'	'itally'	'removing'	'ers'	'data'	'StreamFrame'
'58'	'ometimes'	'providing'	'ed'	'model'	'SolidGoldMagikarp'
'46'	'unbelievably'	'criticizing'	'ation'	'system'	'PayNetMessage'
...

Specifically, there's a cluster containing tokens like 'rot,' 'e,' 'stream,' 'Fame,' 'solid gold Magikarp,' and 'Signet message.' These tokens are quite peculiar and stand out within this embedding cluster. Now, the question arises: What are these tokens, and where do they even come from? For instance, what does 'solid gold Magikarp' mean? It seems nonsensical.

Upon further investigation, they discovered something intriguing. When you ask the language model (LLM) about these tokens—for example, if you request it to repeat back the string 'solid gold Magikarp'—you get a variety of bizarre responses. The LLM's behavior becomes utterly broken. Sometimes it evades the question with a response like 'I'm sorry, I can't hear you.' Other times, it conjures up hallucinations or even delivers insults. It's a fascinating and bewildering phenomenon.

So, you ask it—uh, about the **"streamerbot"**. It tells you, and the model actually just calls you names or it kind of comes up with weird humor. Like, you're actually breaking the model by asking about these very simple strings, like 'attRot' and 'sold gold Magikarp.' So, what the hell is happening? There's a variety of documented behaviors here. There's a bunch of tokens, not just 'sold gold Magikarp,' that exhibit this kind of behavior. Basically, there are trigger words, and if you ask the model about these trigger words or include them in your prompt, the model goes haywire. It exhibits all kinds of really strange behaviors, including some that violate typical safety guidelines and the alignment of the model. It's even swearing back at you. So, what is happening here, and how can this possibly be true? Well, this again comes down to tokenization. What's happening is that 'sold gold Magikarp,' if you actually dig into it, is a Reddit user. There's a user named '[u/SoidlGoldMagikarp](#).' Probably what happened here—although I don't know that this has been definitively explored—is that the tokenization data set was very different from the training data set for the actual language model."

So, in the tokenization data set, there was a ton of Reddit data potentially where the user **Solid Gold Magikarp** was mentioned in the text. Because **Solid Gold Magikarp** was a very common, um, sort of person who would post a lot, this would be a string that occurs many times in a tokenization data set. These tokens would end up getting merged into the single individual token for that single Reddit user, Solid Gold Magikarp. So, they would have a dedicated token in a vocabulary of **50,000 tokens** in GPT-2 that is devoted to that Reddit user.

And then what happens is the tokenization data set has those strings, but later, when you train the model, the language model itself—um, this data from Reddit was not present. Therefore, in the entire training set for the language model, Solid Gold Magikarp never occurs. That token never appears in the training set for the actual language model. Later, this token never gets activated; it's initialized at random at the beginning of optimization.

Then, you have forward and backward passes, as well as updates to the model. However, this token is just never updated in the embedding table. That row vector never gets sampled; it never gets used, so it remains completely untrained. It's akin to unallocated memory in a typical binary program written in C or something similar. At test time, if you evoke this token, you're essentially plucking out a row from the embedding table that is entirely untrained. This feeds into a Transformer and creates undefined behavior. That's what we're seeing here: a completely undefined, never-before-seen behavior during training. So, any of these peculiar tokens would evoke this behavior because fundamentally, the model is out of sample and out of distribution.

The very last thing I wanted to briefly mention is that different formats, representations, and languages might be more or less efficient with GPT tokenizers or any tokenizers for that matter.

So, for example, JSON is actually really dense in tokens, and YAML is a lot more efficient in tokens. So, for example, if we put the same text in JSON and in YAML: the JSON is 116, and the YAML is 99. That's quite a bit of an improvement. In the token economy, where we are paying per token in many ways, you are paying in the context length, and you're paying in the dollar amount for the cost of processing all this kind of structured data. When you have to, it's preferable to use YAML over JSON. In general, the tokenization density is something that you have to care about and worry about at all times. Try to find efficient encoding schemes and spend a lot of time in tiktokenizer, measuring the different token efficiencies of different formats and settings, and so on.

Final recommendations

Okay, so that concludes my fairly long video on tokenization. I know it's tricky, annoying, and sometimes irritating. Personally, I really dislike this stage. However, what I do have to say at this point is: don't brush it off. There are a lot of pitfalls, sharp edges, security issues, and AI safety concerns. For instance, we've seen cases where unallocated memory was plugged into language models. It's worth understanding this stage.

That said, I will also mention that eternal glory goes to anyone who can get rid of it. I showed you one possible paper that attempted to do just that. I hope more research can follow over time.

My final recommendations for applications right now are as follows:

- If you can reuse the GPT-4 tokens and vocabulary in your application, that's something you should seriously consider. Just use the tokenizers library because it is very efficient and a nice choice for inference.
- I'm also a big fan of the byte-level BPE that TikToken and OpenAI use. If, for some reason, you want to train your own vocabulary from scratch, I recommend using BPE with SentencePiece.
- as I mentioned, I'm not a huge fan of SentencePiece. I don't like its byte fallback, and I'm not a fan of how it handles BPE on Unicode code points. It has a million settings, and it's easy to misconfigure them, leading to sentence cropping or other issues. So, be very careful with the settings.

when trying to train yourself, maybe try to copy-paste exactly, maybe where Meta did. Or, basically, spend a lot of time looking at all the hyperparameters and go through the code of SentencePiece. Make sure that you have this correct. But even if you have all the settings correct, I still think that the algorithm is kind of inferior to what's happening here. Maybe the best approach, if you really need to train your vocabulary, is to just wait for MBPE to become as efficient as possible. That's something I hope to work on, and at some point, maybe we can be training something that's essentially what we want: a tokenization code. That is the ideal thing that currently does not exist. MBPE is an implementation of it, but currently, it's in Python. So, that's currently what I have to say about tokenization.