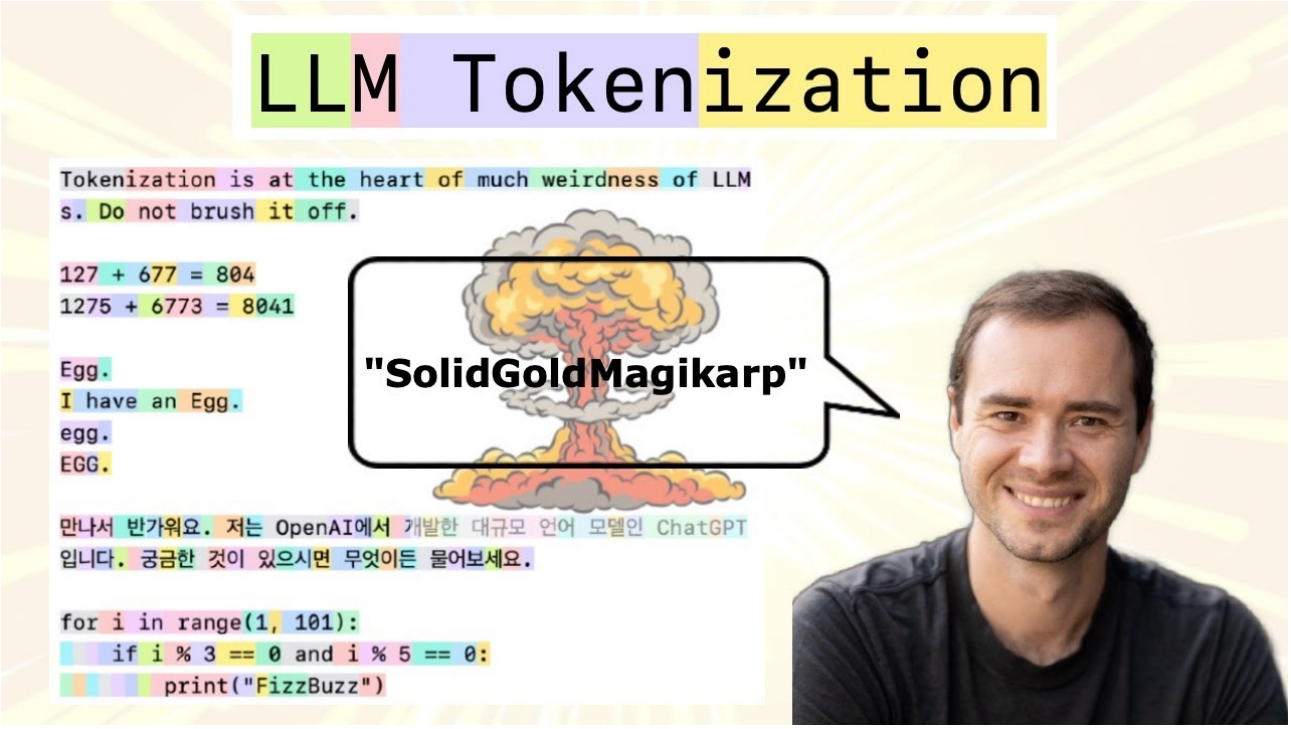
Let's build the GPT Tokenizer



Let's build the GPT Tokenizer

hi everyone so 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 set face and that's because uh tokenization is my least favorite part of working with large language models but unfortunately it is necessary to understand in some detail because it it is fairly hairy gnarly and there's a lot of hidden foot guns to be aware of and a lot of oddness with large language models typically traces back to tokenization so what is tokenization now in my previous video Let's Build GPT from scratch uh we actually already did tokenization but we did a very naive simple version of tokenization so when you go to the Google colab for that video uh you see here that we loaded our training set and our training set was this uh Shakespeare uh data set now in the beginning the Shakespeare data set is just a large string in Python it's just text and so the question is how do we plug text into large language models and in this case here we created a vocabulary of 65 possible characters that we saw occur in this string these were the possible characters and we saw that there are 65 of them and then we created a a lookup table for converting from every possible character a little string piece into a token an integer so here for example we tokenized the string

High there and we received this sequence of tokens and here we took the first 1,000 characters of our data set and we encoded it into tokens and because it is this is character level we received 1,000 tokens in a sequence so token 18 47 Etc now later we saw that the way we plug these tokens into the language model is by using an embedding table and so basically if we have 65 possible tokens then this embedding table is going to have 65 rows and roughly speaking we're taking the integer associated with every single sing Le token we're using that as a lookup into this table and we're plucking out the corresponding row and this row is a uh is trainable parameters that we're going to train using back propagation and this is the vector that then feeds into the Transformer um and that's how the Transformer Ser of perceives every single token so here we had a very naive tokenization process that was a character level tokenizer but in practice in state-ofthe-art uh language models people use a lot more complicated schemes unfortunately uh for constructing these uh token vocabularies so we're not dealing on the Character level we're dealing on chunk level and the way these um character chunks are constructed is using algorithms such as for example the bik pair in coding algorithm which we're going to go into in detail um and cover in this video I'd like to briefly show you the paper that introduced a bite level encoding as a mechanism for tokenization in the context of large language models and I would say that that's probably the gpt2 paper and if you scroll down here to the section input representation this is where they cover tokenization the kinds of properties that you'd like the tokenization to have and they conclude here that they're going to have a tokenizer where you have a vocabulary of 50,2 57 possible tokens and the context size is going to be 1,24 tokens so in the in in the attention layer of the Transformer neural network every single token is attending to the previous tokens in the sequence and it's going to see up to 1,24 tokens so tokens are this like fundamental unit um the atom of uh large language models if you will and everything is in units of tokens everything is about tokens and tokenization is the process for translating strings or text into sequences of tokens and uh 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 get 63 hits um and that's because tokens are again pervasive so here they mentioned that they trained on two trillion tokens of data and so on so we're going to build our own tokenizer luckily the bite be encoding algorithm is not uh that super complicated and we can build it from scratch ourselves and 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 the tokenization because I just want to make sure that we motivate it sufficiently for why we are doing all this and why this is so gross so tokenization is at the heart of a lot of weirdness in large language models and I would advise that you do not brush it off a lot of the issues that may look like just issues

with the new network architecture or the large language model itself are actually issues with the tokenization and fundamentally Trace uh back to it so if you've noticed any issues with large language models can't you know not able to do spelling tasks very easily that's usually due to tokenization simple string processing can be difficult for the large language model to perform natively uh non-english languages can work much worse and to a large extent this is due to tokenization sometimes llms are bad at simple arithmetic also can trace be traced to tokenization uh gbt2 specifically would have had quite a bit more issues with python than uh future versions of it due to tokenization there's a lot of other issues maybe you've seen weird warnings about a trailing whites space this is a tokenization issue um if you had asked GPT earlier about solid gold Magikarp and what it is you would see the llm go totally crazy and it would start going off about a completely unrelated tangent topic maybe you've been told to use yl over Json in structure data all of that has to do with tokenization so basically tokenization is at the heart of many issues I will look back around to these at the end of the video but for now let me just um skip over it a little bit and let's go to this web app um the Tik tokenizer bell.app so I have it loaded here and what I like about this web app is that tokenization is running a sort of live in your browser in JavaScript so you can just type here stuff hello world and the whole string rokenes so here what we see on uh the left is a string that you put in on the right we're currently using the gpt2 tokenizer we see that this string that I pasted here is currently tokenizing into 300 tokens and here they are sort of uh shown explicitly in different colors for every single token so for example uh this word tokenization became two tokens the token 3,642 and 1,634 the token um space is is token 318

so be careful on the bottom you can show white space and keep in mind that there are spaces and uh sln new line characters in here but you can hide them for clarity the token space at is token 379 the to the Token space the is 262 Etc so

you notice here that the space is part of that uh token chunk now so this is kind of like how our English sentence broke up and that seems all well and good now now here I put in some arithmetic so we see that uh the token 127 Plus and then token six space 6 followed by 77 so what's happening here is that 127 is feeding in as a single token into the large language model but the um number 677 will actually feed in as two separate tokens and so the large language model has to sort of um take account of that and process it correctly in its Network and see here 804 will be broken up into two

tokens and it's is all completely arbitrary and here I have another example of four-digit numbers and they break up in a way that they break up and it's totally arbitrary sometimes you have um multiple digits single token sometimes you have individual digits as many tokens and it's all kind of pretty arbitrary and coming out of the tokenizer here's another example we have the string egg and you see here that this became two tokens but for some reason when I say I have an egg you see when it's a space egg it's two token it's sorry it's a single token so just egg by itself in the beginning of a sentence is two tokens but here as a space egg is suddenly a single token uh for the exact same string okay here lowercase egg turns out to be a single token and in particular notice that the color is different so this is a different token so this is case sensitive and of course a capital egg would also be different tokens and again um this would be two tokens arbitrarily so so for the same concept egg depending on if it's in the beginning of a sentence at the end of a sentence lowercase uppercase or mixed all this will be uh basically very 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 training on that these are actually all the exact same concept and it has to sort of group them in the parameters of the neural network and understand just based on the data patterns that these are all very similar but maybe not almost exactly similar but but very very similar um after the EG demonstration here I have um an introduction from open a eyes chbt in Korean so manaso Pang uh Etc uh

so this is in Korean and the reason I put this here is because you'll notice that um non-english languages work slightly worse in Chachi part of this is because of course the training data set for Chachi is much larger for English and for everything else but the same is true not just 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 and what ends up happening is that we're going to have a lot more longer tokens for English so how do I put this if you have a single sentence in English and you tokenize it you might see that it's 10 tokens or something like that but 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 and 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 so you're using up more tokens and then in the attention of the Transformer when these tokens try to attend each other you are running out of context um in the maximum context length of that Transformer and so basically all the non-english text is stretched

out from the perspective of the Transformer and this just has to do with the um trainings that used for the tokenizer and the tokenization itself so it will create a lot bigger tokens and a lot larger groups in English and it will have a lot of little boundaries for all the other non-english text um so if we translated this into English it would be significantly fewer tokens the final example I have here is a little snippet of python for doing FS buuz and what I'd like you to notice is look all these individual spaces are all separate tokens they are token 220 so uh 220 220 220 220 and then space

if is a single token and so what's going on here is that when the Transformer is going to consume or try to uh create this text it needs to um handle all these spaces individually they all feed in one by one into the entire Transformer in the sequence and so this is being extremely wasteful tokenizing it in this way and so as a result of that gpt2 is not very good with python and it's not anything to do with coding or the language model itself it's just that if he use a lot of indentation using space in Python like we usually do uh you just end up bloating out all the text and it's separated across way too much of the sequence and we are running out of the context length in the sequence uh that's roughly speaking what's what's happening we're being way too wasteful we're taking up way too much token space now we can also scroll up here and we can change the tokenizer so note here that gpt2 tokenizer creates a token count of 300 for this string here we can change it to CL 100K base which is the GPT for tokenizer and we see that the token count drops to 185 so for the exact same string we are now roughly having the number of tokens and roughly speaking this is because uh the number of tokens in the GPT 4 tokenizer is roughly double that of the number of tokens in the gpt2 tokenizer so we went went from roughly 50k to roughly 100K now you can imagine that this is a good thing because the same text is now squished into half as many tokens so uh this is a lot denser input to the Transformer and in the Transformer every single token has a finite number of tokens before it that it's going to pay attention to and so what this is doing is we're roughly able to see twice as much text as a context for what token to predict next uh because of this change but of course just increasing the number of tokens is uh not strictly better infinitely uh because as you increase the number of tokens now your embedding table is um sort of getting a lot larger and also at the output we are trying to predict the next token and there's the soft Max there and that grows as well we're going to go into more detail later on this but there's some kind of a Sweet Spot somewhere where you have a just right number of tokens in your vocabulary where everything is appropriately dense and still fairly efficient now one thing I would like you to note specifically for the gp4 tokenizer is that the handling of the white space for python has improved a lot you see that here these four spaces are represented as

one single token for the three spaces here and then the token SPF 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 open aai when they designed the gp4 tokenizer and they group a lot more 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 and so the Improvement in the python coding ability from gbt2 to gp4 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 okay so let's now start writing some code 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 some integers in some fixed vocabulary and then we will use those integers to make a look up into a lookup table of vectors and feed those vectors into the Transformer as an input now the reason this gets a little bit tricky of course is that we don't just want to support the simple English alphabet we want to support different kinds of languages so this is anango in Korean which is hello and we also want to support many kinds of special characters that we might find on the internet for example Emoji so how do we feed this text into uh Transformers well how's the what is this text anyway in Python so if you go to the documentation of a string in Python you can see that strings are immutable sequences of Unicode code points okay what are Unicode code points we can go to PDF so Unicode code points are defined by the Unicode Consortium as part of the Unicode standard and what this is really is that it's just a definition of roughly 150,000 characters right now and roughly speaking what they look like and what integers um represent those characters so it says 150,000 characters across 161 scripts as of right now so if you scroll down here you can see that the standard is very much alive the latest standard 15.1 in September 2023 and basically this is just a way to define lots of types of characters like for example all these characters across different scripts so the way we can access the unic code code Point given Single Character is by using the or function in Python so for example I can pass in Ord of H and I can see that for the Single Character H the unic code code point is 104 okay um but this can be arbitr complicated so we can take for example our Emoji here and we can see that the code point for this one is 128,000 or we can take un and this is 50,000 now keep in mind you can't plug in strings here because you uh this doesn't have a single code point it only takes a single uni code code Point character and tells you its integer so in this way we can look up all the um characters of this specific string and their code points so or of X forx in this string and we get this encoding here now see here we've already turned the raw code points already have integers so why can't we simply just use these integers and

not have any tokenization at all why can't we just use this natively as is and just use the code Point well one reason for that of course is that the vocabulary in that case would be quite long so in this case for Unicode the this is a vocabulary of 150,000 different code points but more worryingly than that I think the Unicode standard is very much alive and it keeps changing and so it's not kind of a stable representation necessarily that we may want to use directly so for those reasons we need something a bit better so to find something better we turn to encodings so if we go to the Wikipedia page here we see that the Unicode consortion defines three types of encodings utf8 UTF 16 and UTF 32 these encoding are the way by which we can take Unicode text and translate it into binary data or by streams utf8 is by far the most common uh so this is the utf8 page now this Wikipedia page is actually quite long but what's important for our purposes is that utf8 takes every single Cod point and it translates it to a by stream and this by stream is between one to four bytes so it's a variable length encoding so depending on the Unicode Point according to the schema you're going to end up with between 1 to four bytes for each code point on top of that there's utf8 uh utf16 and UTF 32 UTF 32 is nice because it is fixed length instead of variable length but it has many other downsides as well so the full kind of spectrum of pros and cons of all these different three encodings are beyond the scope of this video I just like to point out that I enjoyed this block post and this block post at the end of it also has a number of references that can be quite useful uh one of them is uh utf8 everywhere Manifesto um and this Manifesto describes the reason why utf8 is significantly preferred and a lot nicer than the other encodings and why it is used a lot more prominently um on the internet one of the major advantages just just to give you a sense is that utf8 is the only one of these that is backwards compatible to the much simpler asky encoding of text um but I'm not going to go into the full detail in this video so suffice to say that we like the utf8 encoding and uh let's try to take the string and see what we get if we encoded into utf8 the string class in Python actually has do encode and you can give it the encoding which is say utf8 now we get out of this is not very nice because this is the bytes is a bytes object and it's not very nice in the way that it's printed so I personally like to take it through list because then we actually get the raw B of this uh encoding so this is the raw byes that represent this string according to the utf8 en coding we can also look at utf16 we get a slightly different by stream and we here we start to see one of the disadvantages of utf16 you see how we have zero Z something Z something Z something we're starting to get a sense that this is a bit of a wasteful encoding and indeed for simple asky characters or English characters here uh we just have the structure of 0 something Z something and it's not exactly nice same for UTF 32 when we expand this we can start to get a sense of the wastefulness of this encoding for our purposes you see a lot of zeros followed by

something and so uh this is not desirable so suffice it to say that we would like to stick with utf8 for our purposes however if we just use utf8 naively these are by streams so that would imply a vocabulary length of only 256 possible tokens uh but this this vocabulary size is very very small what this is going to do if we just were to use it naively is that all of our text would be stretched out over very very long sequences of bytes and so um what what this does is that certainly the embeding table is going to be tiny and the prediction at the top at the final layer is going to be very tiny but our sequences are very long and remember that we have pretty finite um context length and the attention that we can support in a transformer for computational reasons and so we only have as much context length but now we have very very long sequences and this is just inefficient and it's not going to allow us to attend to sufficiently long text uh before us for the purposes of the next token prediction task so we don't want to use the raw bytes of the utf8 encoding we want to be able to support larger vocabulary size that we can tune as a hyper but we want to stick with the utf8 encoding of these strings so what do we do well the answer of course is we turn to the bite pair encoding algorithm which will allow us to compress these bite sequences um to a variable amount so we'll get to that in a bit but I just want to briefly speak to the fact that I would love nothing more than to be able to feed raw bite sequences into uh language models in fact there's a paper about how this could potentially be done uh from Summer last last year now the problem is you actually have to go in and you have to modify the Transformer architecture because as I mentioned you're going to have a problem where the attention will start to become extremely expensive because the sequences are so long and so in this paper they propose kind of a hierarchical structuring of the Transformer that could allow you to just feed in raw bites and so at the end they say together these results establish the viability of tokenization free autor regressive sequence modeling at scale so tokenization free would indeed be amazing we would just feed B streams directly into our models but unfortunately I don't know that this has really been proven out yet by sufficiently many groups and a sufficient scale uh but something like this at one point would be amazing and I hope someone comes up with it but for now we have to come back and we can't feed this directly into language models and we have to compress it using the B paare encoding algorithm so let's see how that works so as I mentioned the B paare encoding algorithm is not all that complicated and the Wikipedia page is actually quite instructive as far as the basic idea goes go what we're doing is we have some kind of a input sequence uh like for example here we have only four elements in our vocabulary a b c and d and we have a sequence of them so instead of bytes let's say we just have four a vocab size of four the sequence is too long and we'd like to compress it so what we do is that we iteratively find the pair of uh tokens that occur the most frequently and then

once we've identified that pair we repl replace that pair with just a single new token that we append to our vocabulary so for example here the bite pair AA occurs most often so we mint a new token let's call it capital Z and we replace every single occurrence of AA by Z so now we have two Z's here so here we took a sequence of 11 characters with vocabulary size four and we've converted it to a um sequence of only nine tokens but now with a vocabulary of five because we have a fifth vocabulary element that we just created and it's Z standing for concatination of AA and we can again repeat this process so we again look at the sequence and identify the pair of tokens that are most frequent let's say that that is now AB well we are going to replace AB with a new token that we meant call Y so y becomes ab and then every single occurrence of ab is now replaced with y so we end up with this so now we only have 1 2 3 4 5 6 seven characters in our sequence but we have not just um four vocabulary elements or five but now we have six and for the final round we again look through the sequence find that the phrase zy or the pair zy is most common and replace it one more time with another um character let's say x so X is z y and we replace all curses of zy and we get this following sequence so basically after we have gone through this process instead of having a um sequence of 11 uh tokens with a vocabulary length of four we now have a sequence of 1 2 3 four five tokens but our vocabulary length now is seven and so in this way we can iteratively compress our sequence I we Mint new tokens so in the in the exact same way we start we start out with bite sequences so we have 256 vocabulary size but we're now going to go through these and find the bite pairs that occur the most and we're going to iteratively start minting new tokens appending them to our vocabulary and replacing things and in this way we're going to end up with a compressed training data set and also an algorithm for taking any arbitrary sequence and encoding it using this uh vocabul and also decoding it back to Strings so let's now Implement all that so here's what I did I went to this block post that I enjoyed and I took the first paragraph and I copy pasted it here into text so this is one very long line here now to get the tokens as I mentioned we just take our text and we encode it into utf8 the tokens here at this point will be a raw bites single stream of bytes and just so that it's easier to work with instead of just a bytes object I'm going to convert all those bytes to integers and then create a list of it just so it's easier for us to manipulate and work with in Python and visualize and here I'm printing all of that so this is the original um this is the original paragraph and its length is 533 uh code points and then here are the bytes encoded in ut utf8 and we see that this has a length of 616 bytes at this point or 616 tokens and the reason this is more is because a lot of these simple asky characters or simple characters they just become a single bite but a lot of these Unicode more complex characters become multiple bytes up to four and so we are expanding that size so now what we'd like to do as

a first step of the algorithm is we'd like to iterate over here and find the pair of bites that occur most frequently because we're then going to merge it so if you are working long on a notebook on a side then I encourage you to basically click on the link find this notebook and try to write that function yourself otherwise I'm going to come here and 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 basically the counts and then this is a pythonic way to iterate consecutive elements of this list uh which we covered in the previous video and then here I'm just keeping track of just incrementing by one um for all the pairs so if I call this on all the tokens here then the stats comes out here so this is the dictionary the keys are these topples of consecutive elements and this is the count so just to uh print it in a slightly better way this is one way that I like to do that where you it's a little bit compound here so you can pause if you like but we iterate all all the items the items called on dictionary returns pairs of key value and instead I create a list here of value key because if it's a value key list then I can call sort on it and by default python will uh use the first element which in this case will be value to sort by if it's given tles and then reverse so it's descending and print that so basically it looks like 101 comma 32 was the most commonly occurring consecutive pair and it occurred 20 times we can double check that that makes reasonable sense so if I just search 10132 then you see that these are the 20 occurrences of that um pair and if we'd like to take a look at what exactly that pair is we can use Char which is the opposite of or in Python so we give it a um unic code Cod point so 101 and of 32 and we see that this is e and space so basically there's a lot of E space here meaning that a lot of these words seem to end with e so here's eace as an example so there's a lot of that going on here and this is the most common pair so now that we've identified the most common pair we would like to iterate over this sequence we're going to Mint a new token with the ID of 256 right because these tokens currently go from Z to 255 so when we create a new token it will have an ID of 256 and we're going to iterate over this entire um list and every every time we see 101 comma 32 we're going to swap that out for 256 so let's Implement that now and feel free to uh do that yourself as well so first I commented uh this just so we don't pollute uh the notebook too much this is a nice way of in Python obtaining the highest ranking pair so we're basically calling the Max on this dictionary stats and this will return the maximum key and then the question is how does it rank keys so you can provide it with a function that ranks keys and that function is just stats. getet uh stats. getet would basically return the value and so we're ranking by the value and getting the maximum key so it's 101 comma 32 as we saw now to actually merge 10132 um this is the function that I wrote but again there are many different versions

of it so we're going to take a list of IDs and the the pair that we want to replace and that pair will be replaced with the new index idx so iterating through IDs if we find the pair swap it out for idx so we create this new list and then we start at zero and then we go through this entire list sequentially from left to right and here we are checking for equality at the current position with the pair um so here we are checking that the pair matches now here is a bit of a tricky condition that you have to append if you're trying to be careful and that is that um you don't want this here to be out of Bounds at the very last position when you're on the rightmost element of this list otherwise this would uh give you an autof bounds error so we have to make sure that we're not at the very very last element so uh this would be false for that so if we find a match we append to this new list that replacement index and we increment the position by two so we skip over that entire pair but otherwise if we we haven't found a matching pair we just sort of copy over the um element at that position and increment by one then return this so 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 then calling this on that will give us what we're asking for so here the 67 is replaced with 99 so now I'm going to uncomment this for our actual use case where we want to take our tokens we want to take the top pair here and replace it with 256 to get tokens to if we run this we get the following so recall that previously we had a length 616 in this list and now we have a length 596 right so this decreased by 20 which makes sense because there are 20 occurrences moreover we can try to find 256 here and we see plenty of occurrences on off it and moreover just double check there should be no occurrence of 10132 so this is the original array plenty of them and in the second array there are no occurrences of 1032 so we've successfully merged this single pair and now we just uh iterate this so we are going to go over the sequence again find the most common pair and replace it so let me now write a y Loop that uses these functions to do this um sort of iteratively and how many times do we do it four well that's totally up to us as a hyper parameter the more um steps we take the larger will be our vocabulary and the shorter will be our sequence and there is some sweet spot that we usually find works the best in practice and so this is kind of a hyperparameter and we tune it and we find good vocabulary sizes as an example gp4 currently uses roughly 100,000 tokens and um bpark that those are reasonable numbers currently instead the are large language models so let me now write uh putting putting it all together and uh iterating these steps okay now before we dive into the Y loop I wanted to add one more cell here where I went to the block post and instead of grabbing just the first paragraph or two I took the entire block post and I stretched it out in a single line and basically just using longer text will allow us to have more representative statistics for the bite Pairs and we'll just get a more sensible results out of it because it's longer text um so

here we have the raw text we encode it into bytes using the utf8 encoding and then here as before we are just changing it into a list of integers in Python just so it's easier to work with instead of the raw byes objects and then this is the code that I came up with uh to actually do the merging in Loop these two functions here are identical to what we had above I only included them here just so that you have the point of reference here so uh these two are identical and then this is the new code that I added so the first first thing we want to do is we want to decide on the final vocabulary size that we want our tokenizer to have and as I mentioned this is a hyper parameter and you set it in some way depending on your best performance so let's say for us we're going to use 276 because that way we're going to be doing exactly 20 merges and uh 20 merges because we already have 256 tokens for the raw bytes and to reach 276 we have to do 20 merges uh to add 20 new tokens here uh this is uh one way in Python to just create a copy of a list 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 so this is just a copy operation then here I'm creating a merges uh dictionary so this merges dictionary is going to maintain basically the child one child two mapping to a new uh token and so what we're going to be building up here is a binary tree of merges but actually 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 bites those are the starting 256 tokens and then we're starting to like merge two of them at a time and so it's not a tree it's more like a forest um uh as we merge these elements so for 20 merges we're going to find the most commonly occurring pair we're going to Mint a new token integer for it so I here will start at zero so we'll going to start at 256 we're going to print that we're merging it and we're going to replace all of the occurrences of that pair with the new new lied token and we're going to record that this pair of integers merged into this new integer so running this gives us the following output so we did 20 merges and for example the first merge was exactly as before the 10132 um tokens merging into a new token 2556 now keep in mind that the individual uh tokens 101 and 32 can still occur in the sequence after merging it's only when they occur exactly consecutively that that becomes 256 now um and in particular the other thing to notice here is that the token 256 which is the newly minted token is also eligible for merging so here on the bottom 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 ration so 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 as well is we can take a look at the compression ratio that we've achieved so in particular we started off with this tokens list um so we started off with 24,000 bytes and after merging 20 times uh we now have only 19,000 um tokens and so therefore the compression ratio

simply just dividing the two is roughly 1.27 so that's the amount of compression we were able to achieve of this text with only 20 merges um and of course the more vocabulary elements you add uh the greater the compression ratio here would be finally so that's kind of like um the training of the tokenizer if you will now 1 Point I wanted to make is that and maybe this is a diagram that can help um kind of illustrate is that tokenizer is a completely separate object from the large language model itself so everything in this lecture we're not really touching the llm itself uh we're just training the tokenizer this is a completely separate pre-processing stage usually so the tokenizer will have its own training set just like a large language model has a potentially different training set so the tokenizer has a training set of documents on which you're going to train the tokenizer and then and um we're performing The Bite pair encoding algorithm as we saw above to train the vocabulary of this tokenizer so it has its own training set it is a pre-processing stage that you would run a single time in the beginning um and the tokenizer is trained using bipar coding algorithm once you have the tokenizer once it's trained and you have the vocabulary and you have the merges uh we can do both encoding and decoding so these two arrows here so the tokenizer is 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 so now that we have trained uh tokenizer and we have these merges we are going to turn to how we can do the encoding and the decoding step if you give me text here are the tokens and vice versa if you give me tokens here's the text once we have that we can translate between these two Realms and then the language model is going to be trained as a step two afterwards and typically in a in a sort of a state-of-the-art application you might take all of your training data for the language model and you might run it through the tokenizer and sort of translate everything into a massive token sequence and then you can throw away the raw text you're just left with the tokens themselves and those are stored on disk and that is what the large language model is actually reading when it's training on them so this one approach that you can take as a single massive pre-processing step a stage um so yeah basically I think the most important thing I want to get across is that this is completely separate stage it usually has its own entire uh training set you may want to have those training sets be different between the tokenizer and the logge language model so for example when you're training the tokenizer as I mentioned we don't just care about the performance of English text we care about uh multi many different languages and we also care about code or not code so you may want to look into different kinds of mixtures of different kinds of languages and different amounts of code and things like that because the amount of different language that you have in your tokenizer training set will determine

how many merges of it there will be and therefore that determines the density with which uh this type of data is um sort of has in the token space and so roughly speaking intuitively if you add some amount of data like say you have a ton of Japanese data in your uh tokenizer training set then that means that more Japanese tokens will get merged and therefore Japanese will have shorter sequences uh and that's going to be beneficial for the large language model which has a finite context length on which it can work on in in the token space uh so hopefully that makes sense so we're now going to turn to encoding and decoding now that we have trained a tokenizer so we have our merges and now how do we do encoding and decoding okay so let's begin with decoding which is this Arrow over here so given a token sequence let's go through the tokenizer to get back a python string object so the raw text so this is the function that we' like to implement um we're given the list of integers and we want to return a python string if you'd like uh try to implement this function yourself it's a fun exercise otherwise I'm going to start uh pasting in my own solution so there are many different ways to do it um here's one way I will create an uh kind of pre-processing variable that I will call vocab and vocab is a mapping or a dictionary in Python for from the token uh ID to the bytes object for that token so we begin with the raw bytes for tokens from 0 to 255 and then we go in order of all the merges and we sort of uh populate this vocab list by doing an addition here so this is the basically the bytes representation of the first child followed by the second one and remember these are bytes objects so this addition here is an addition of two bytes objects just concatenation so that's what we get here one tricky thing to be careful with by the way is that I'm iterating a dictionary in Python using a DOT items and uh it really matters that this runs in the order in which we inserted items into the merous 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 but we are using an um modern python so we're okay and then here uh given the IDS the first thing we're going to do is get the tokens so the way I implemented this here is I'm taking I'm iterating over all the IDS I'm using vocap to look up their bytes and then here this is one way in Python to concatenate all these bytes together to create our tokens and then these tokens here at this point are raw bytes so I have to decode using UTF F now back into python strings so previously we called that encode on a string object to get the bytes and now we're doing it Opposite we're taking the bytes and calling a decode on the bytes object to get a string in Python and then we can return text so um this is how we can do it now this actually has a um issue um in the way I implemented it and this could actually throw an error so try to think figure out why this code could actually result in an error if we plug in um uh some sequence of IDs that is unlucky so let me demonstrate

the issue when I try to decode just something like 97 I am going to get letter A here back so nothing too crazy happening but when I try to decode 128 as a single element the token 128 is what in string or in Python object uni Cod decoder utfa can't Decode by um 0x8 which is this in HEX in position zero invalid start bite what does that mean well to understand what this means we have to go back to our utf8 page uh that I briefly showed earlier and this is Wikipedia utf8 and basically there's a specific schema that utfa bytes take so in particular if you have a multi-te object for some of the Unicode characters they have to have this special sort of envelope in how the encoding works and so what's happening here is that invalid start pite that's because 128 the binary representation of it is one followed by all zeros so we have one and then all zero and we see here that that doesn't conform to the format because one followed by all zero just doesn't fit any of these rules so to speak so it's an invalid start bite which is byte one this one must have a one following it and then a zero following it and then the content of your uni codee in x here so basically we don't um exactly follow the utf8 standard and this cannot be decoded and so the way to fix this um is to use this errors equals in bytes. decode function of python and by default errors is strict so we will throw an error if um it's not valid utf8 bytes encoding but there are many different things that you could put here on error handling this is the full list of all the errors that you can use and in particular instead of strict let's change it to replace and that will replace uh with this special marker this replacement character so errors equals replace and now we just get that character back so basically not every single by sequence is valid utf8 and if it happens that your large language model for example predicts your tokens in a bad manner then they might not fall into valid utf8 and then we won't be able to decode them so the standard practice is to basically uh use errors equals replace and this is what you will also find in the openai um code that they released as well but basically whenever you see um this kind of a character in your output in that case uh something went wrong and the LM output not was not valid uh sort of sequence of tokens okay and now we're going to go the other way so we are going to implement this Arrow right here where we are going to be given a string and we want to encode it into tokens so this is the signature of the function that we're interested in and um this should basically print a list of integers of the tokens so again uh try to maybe implement this yourself if you'd like a fun exercise uh and pause here otherwise I'm going to start putting in my solution so again there are many ways to do this so um this is one of the ways that sort of I came came up with so the first thing we're going to do is we are going to uh take our text encode it into utf8 to get the raw bytes and then as before we're going to call list on the bytes object to get a list of integers of those bytes so those are the starting tokens those are the raw bytes of our sequence but now of course according to the merges dictionary

above and recall this was the merges some of the bytes may be merged according to this lookup in addition to that remember that the merges was built from top to bottom and this is sort of the order in which we inserted stuff into merges and so we prefer to do all these merges in the beginning before we do these merges later because um for example this merge over here relies on the 256 which got merged here so we have to go in the order from top to bottom sort of if we are going to be merging anything now we expect to be doing a few merges so we're going to be doing W true um and now we want to find a pair of byes that is consecutive that we are allowed to merge according to this in order to reuse some of the functionality that we've already written I'm going to reuse the function uh get stats so recall that get stats uh will give us the we'll basically count up how many times every single pair occurs in our sequence of tokens and return that as a dictionary and the dictionary was a mapping from all the different uh by pairs to the number of times that they occur right um at this point we don't actually care how many times they occur in the sequence we only care what the raw pairs are in that sequence and so I'm only going to be using basically the keys of the dictionary I only care about 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 so what do we want we want to find the pair or like the a key inside stats that has the lowest index in the merges uh dictionary because we want to do all the early merges before we work our way to the late merges so again there are many different ways to implement this but I'm going to do something a little bit fancy here so I'm going to be using the Min over an iterator in Python when you call Min on an iterator and stats here as a dictionary we're going to be iterating the keys of this dictionary in Python so we're looking at all the pairs inside stats um which are all the consecutive Pairs and we're going to be taking the consecutive pair inside tokens that has the minimum what the Min takes a key which gives us the function that is going to return a value over which we're going to do the Min and the one we care about is we're we care about taking merges and basically getting um that pairs index so basically for any pair inside stats we are going to be looking into merges at what index it has and we want to get the pair with the Min number so as an example if there's a pair 101 and 32 we definitely want to get that pair uh we want to identify it here and return it and pair would become 10132 if it occurs and the reason that I'm putting a float INF here as a fall back is that in the get function when we call uh when we basically consider a pair that doesn't occur in the merges then that pair is not eligible to be merged right so if in the token sequence there's some pair that is not a merging pair it cannot be merged then uh it doesn't actually occur here and it doesn't have an index and uh it cannot be merged which we will denote as float INF and the reason Infinity is nice here is because for sure we're guaranteed that it's not going to

participate in the list of candidates when we do the men so uh so this is one way to do it so B basically long story short this Returns the most eligible merging candidate pair uh that occurs in the tokens now one thing to be careful with here is this uh function here might fail in the following way if there's nothing to merge then uh uh then there's nothing in merges um that satisfi that is satisfied anymore there's nothing to merge everything just returns float imps and then the pair I think will just become the very first element of stats um but this pair is not actually a mergeable pair it just becomes the first pair inside stats arbitrarily because all of these pairs evaluate to float in for the merging Criterion so basically it could be that this this doesn't look succeed because there's no more merging pairs so if this pair is not in merges that was returned then this is a signal for us that actually there was nothing to merge no single pair can be merged anymore in that case we will break out um nothing else can be merged you may come up with a different implementation by the way this is kind of like really trying hard in Python um but really we're just trying to find a pair that can be merged with the lowest index here now if we did find a pair that is inside merges with the lowest index then we can merge it so we're going to look into the merger dictionary for that pair to look up the index and we're going to now merge that into that index so we're going to do tokens equals and we're going to replace the original tokens we're going to be replacing the pair pair and we're going to be replacing it with index idx and this returns a new list of tokens where every occurrence of pair is replaced with idx so we're doing a merge and we're going to be continuing this until eventually nothing can be merged we'll come out here and we'll break out and here we just return tokens and so that that's the implementation I think so hopefully this runs okay cool um yeah and this looks uh reasonable so for example 32 is a space in asky so that's here um so this looks like it worked great okay so let's wrap up this section of the video at least I wanted to point out that this is not quite the right implementation just yet because we are leaving out a special case so in particular if uh we try to do this this would give us an error and the issue is that um if we only have a single character or an empty string then stats is empty and that causes an issue inside Min so one way to fight this is if L of tokens is at least two because if it's less than two it's just a single token or no tokens then let's just uh there's nothing to merge so we just return so that would fix uh that case Okay and then second I have a few test cases here for us as well so first let's make sure uh about or let's note the following if we take a string and we try to encode it and then decode it back you'd expect to get the same string back right is that true for all strings so I think uh so here it is the case and I think in general this is probably the case um but notice that going backwards is not is not you're not going to have an identity going backwards because as I mentioned us not all token sequences are valid utf8 uh sort of by streams and so so therefore you're some of them

can't even be decodable um so this only goes in One Direction but for that one direction we can check uh here if we take the training text which is the text that we train to tokenizer around we can make sure that when we encode and decode we get the same thing back which is true and here I took some validation data so I went to I think this web page and I grabbed some text so this is text that the tokenizer has not seen and we can make sure that this also works um okay so that gives us some confidence that this was correctly implemented so those are the basics of the bite pair encoding algorithm we saw how we can uh take some training set train a tokenizer the parameters of this tokenizer really are just this dictionary of merges and that basically creates the little binary Forest on top of raw bites once we have this the merges table we can both encode and decode between raw text and token sequences so that's the the simplest setting of The tokenizer what we're going to do now though is we're going to look at some of the St the art lar language models and the kinds of tokenizers that they use and we're going to see that this picture complexifies very quickly so we're going to go through the details of this comp complexification one at a time so let's kick things off by looking at the GPD Series so in particular I have the gpt2 paper here um and this paper is from 2019 or so so 5 years ago and let's scroll down to input representation this is where they talk about the tokenizer that they're using for gpd2 now this is all fairly readable so I encourage you to pause and um read this yourself but this is where they motivate the use of the bite pair encoding algorithm on the bite level representation of utf8 encoding so this is where they motivate it and they talk about the vocabulary sizes and everything now everything here is exactly as we've covered it so far but things start to depart around here so what they mention is that they don't just apply the naive algorithm as we have done it and in particular here's a example suppose that you have common words like dog what will happen is that dog of course occurs very frequently in the text and it occurs right next to all kinds of punctuation as an example so doc dot dog exclamation mark dog question mark Etc and naively you might imagine that the BP algorithm could merge these to be single tokens and then you end up with lots of tokens that are just like dog with a slightly different punctuation and so it feels like you're clustering things that shouldn't be clustered you're combining kind of semantics with uation and this uh feels suboptimal and indeed they also say that this is suboptimal according to some of the experiments so what they want to do is they want to top down in a manual way enforce that some types of um characters should never be merged together um so they want to enforce these merging rules on top of the bite PA encoding algorithm so let's take a look um at their code and see how they actually enforce this and what kinds of mergy they actually do perform so I have to to tab open here for gpt2 under open AI on GitHub and when we go to Source there is an encoder thatp now I don't personally love that

they call it encoder dopy because this is the tokenizer and the tokenizer can do both encode and decode uh so it feels kind of awkward to me that it's called encoder but that is the tokenizer and there's a lot going on here and we're going to step through it in detail at one point for now I just want to focus on this part here the create a rigix pattern here that looks very complicated and we're going to go through it in a bit uh but this is the core part that allows them to enforce rules uh for what parts of the text Will Never Be merged for sure now notice that re. compile here is a little bit misleading because we're not just doing import re which is the python re module we're doing import reex as re and reex is a python package that you can install P install r x and it's basically an extension of re so it's a bit more powerful re um so let's take a look at this pattern and what it's doing and why this is actually doing the separation that they are looking for okay so I've copy pasted the pattern here to our jupit notebook where we left off and let's take this pattern for a spin so in the exact same way that their code does we're going to call an re. findall for this pattern on any arbitrary string that we are interested so this is the string that we want to encode into tokens um to feed into n llm like gpt2 so what exactly is this doing well re. findall will take this pattern and try to match it against a string um the way this works is that you are going from left to right in the string and you're trying to match the pattern and R.F find all will get all the occurrences and organize them into a list now when you look at the um when you look at this pattern first of all notice that this is a raw string um and then these are three double quotes just to start the string so really the string itself this is the pattern itself right and notice that it's made up of a lot of ores so see these vertical bars those are ores in reg X and so you go from left to right in this pattern and try to match it against the string wherever you are so we have hello and we're going to try to match it well it's not apostrophe s it's not apostrophe t or any of these but it is an optional space followed by- P of uh sorry SL P of L one or more times what is/ P of L it is coming to some documentation that I found um there might be other sources as well uh SLP is a letter any kind of letter from any language and hello is made up of letters h e l Etc so optional space followed by a bunch of letters one or more letters is going to match hello but then the match ends because a white space is not a letter so from there on begins a new sort of attempt to match against the string again and starting in here we're going to skip over all of these again until we get to the exact same Point again and we see that there's an optional space this is the optional space followed by a bunch of letters one or more of them and so that matches so when we run this we get a list of two elements hello and then space world so how are you if we add more letters we would just get them like this now what is this doing and why is this important we are taking our string and instead of directly encoding it um for tokenization we are first splitting it up and when you actually step through the code and we'll do that in a bit more

detail what really is doing on a high level is that it first splits your text into a list of texts just like this one and all these elements of this list are processed independently by the tokenizer and all of the results of that processing are simply concatenated so hello world oh I I missed how hello world how are you we have five elements of list all of these will independent 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 in individually and um after you've done all the possible merging for all of these elements individually the results of all that will be joined um by concatenation and so you are basically what what you're doing effectively is you are never going to be merging this e with this space because they are now parts of the separate elements of this list and so you are saying we are never going to merge eace um because we're breaking it up in this way so basically using this regx pattern to Chunk Up the text is just one way of enforcing that some merges are not to happen and 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 we're trying to not merge across letters across numbers across punctuation and so on so let's see in more detail how that works so let's continue now we have/ P ofn if you go to the documentation SLP of n is any kind of numeric character in any script so it's numbers so we have an optional space followed by numbers and those would be separated out so letters and numbers are being separated so if I do Hello World 123 how are you then world will stop matching here because one is not a letter anymore but one is a number so this group will match for that and we'll get it as a separate entity uh let's see how these apostrophes work so here if we have um uh Slash V or I mean apostrophe V as an example then apostrophe here is not a letter or a number so hello will stop matching and then we will exactly match this with that so that will come out as a separate thing so why are they doing the apostrophes here honestly I think that these are just like very common apostrophes p uh that are used um typically I don't love that they've done this because uh let me show you what happens when you have uh some Unicode apostrophes like for example you can have if you have house then this will be separated out because of this matching but if you use the Unicode apostrophe like this then suddenly this does not work and so this apostrophe will actually become its own thing now and so so um it's basically hardcoded for this specific kind of apostrophe and uh otherwise they become completely separate tokens in addition to this you can go to the gpt2 docs and here when they Define the pattern they say should have added re. ignore case so BP merges can happen for capitalized versions of contractions so what they're pointing out is that you see how this is apostrophe and then lowercase letters well because they didn't

do re. ignore case then then um these rules will not separate out the apostrophes if it's uppercase so house would be like this but if I did

house if I'm uppercase then notice suddenly the apostrophe comes by itself so the tokenization will work differently in uppercase and lower case inconsistently separating out these apostrophes so it feels extremely gnarly and slightly gross um but that's that's how that works okay so let's come back after trying to match a bunch of apostrophe Expressions by the way the other issue here is that these are quite language specific probably so I don't know that all the languages for example use or don't use apostrophes but that would be inconsistently tokenized as a result then we try to match letters then we try to match numbers and then if that doesn't work we fall back to here and what this is saying is again optional space followed by something that is not a letter number or a space in one or more of that so what this is doing effectively is this is trying to match punctuation roughly speaking not letters and not numbers so this group will try to trigger for that so if I do something like this then these parts here are not letters or numbers but they will actually they are uh they will actually get caught here and so they become its own group so we've separated out the punctuation and finally this um this is also a little bit confusing so this is matching white space but this is using a negative look ahead assertion in regex so what this is doing is it's matching wh space up to but not including the last Whit space character why is this important um this is pretty subtle I think so you see how the white space is always included at the beginning of the word so um space r space u Etc suppose we have a lot of spaces here what's going to happen here is that these spaces up to not including the last character will get caught by this and what that will do is it will separate out the spaces up to but not including the last character so that the last character can come here and join with the um space you and 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 view but now we have all this extra white space so basically the GB to tokenizer really likes to have a space letters or numbers um and it it preens these spaces and this is just something that it is consistent about so that's what that is for and then finally we have all the the last fallback is um whites space characters uh so um that would be just um if that doesn't get caught then this thing will catch any trailing spaces and so on I wanted to show one more real world example here so 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 so you'll notice that the list has many elements here and that's because we are splitting up fairly often uh every time sort of a category changes um so there

will never be any merges Within These elements and um that's what you are seeing here now you might think that in order to train the tokenizer uh open AI has used this to split up text into chunks and then run just a BP 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 uh those Spaces end up being entire elements but these spaces never actually end up being merged by by open Ai and the way you can tell is that if you copy paste the exact same chunk here into Tik token U Tik tokenizer you see that all the spaces are kept independent and they're all token 220 so I think opena at some point Point en Force some rule that these spaces would never be merged and so um there's some additional rules on top of just chunking and bpe that open ey is not uh clear about now the training code for the gpt2 tokenizer was never released so all we have is uh 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 training tokenizer this is just the inference code which Tak takes the merges that we have up above and applies them to a new piece of text and so we don't know exactly how opening ey trained um train the tokenizer but it wasn't as simple as chunk it up and BP it uh whatever it was next I wanted to introduce you to the Tik token library from openai which is the official library for tokenization from openai so this is Tik token bip install P to Tik token and then um you can do the tokenization in inference this is again not training code this is only inference code for tokenization um I wanted to show you how you would use it quite simple and running this just gives us the gpt2 tokens or the GPT 4 tokens so this is the tokenizer use for GPT 4 and so in particular we see that the Whit space in gpt2 remains unmerged but in GPT 4 uh these Whit spaces merge as we also saw in this one where here they're all unmerged but if we go down to GPT 4 uh they become merged um now in the gp4 uh tokenizer they changed the regular expression that they use to Chunk Up text so the way to see this is that if you come to your the Tik token uh library and then you go to this file Tik token X openi public this is where sort of like the definition of all these different tokenizers that openi maintains is and so uh necessarily to do the inference they had to publish some of the details about the strings so this is the string that we already saw for gpt2 it is slightly different but it is actually equivalent uh to what we discussed here so this pattern that we discussed is equivalent to this pattern this one just executes a little bit faster so here you see a little bit of a slightly different definition but otherwise it's the same we're going to go into special tokens in a bit and then if you scroll down to CL 100k this is the GPT 4 tokenizer you see that the pattern has changed um and this is kind of like the main the major change in addition to a bunch of other special tokens which I'll go into in a bit again now some I'm not going to actually go into the full detail of the pattern change because honestly this is my

numbing uh I would just advise that you pull out chat GPT and the regex documentation and just step through it but really the major changes are number one you see this eye here that means that the um case sensitivity this is case insensitive match and so the comment that we saw earlier on oh we should have used re. uppercase uh basically we're now going to be matching these apostrophe s apostrophe D apostrophe M Etc uh we're going to be matching them both in lowercase and in uppercase so that's fixed there's a bunch of different like handling of the whites space that I'm not going to go into the full details of and then one more thing here is you will notice that when they match the numbers they only match one to three numbers so so they will never merge numbers that are in low in more than three digits only up to three digits of numbers will ever be merged and uh that's one change that they made as well to prevent uh tokens that are very very long number sequences uh but again we don't really know why they do any of this stuff uh because none of this is documented and uh it's just we just get the pattern so um yeah it is what it is but those are some of the changes that gp4 has made and of course the vocabulary size went from roughly 50k to roughly 100K the next thing I would like to do very briefly is to take you through the gpt2 encoder dopy that openi has released uh this is the file that I already mentioned to you briefly now this file is uh fairly short and should be relatively understandable to you at this point um starting at the bottom here they are loading two files encoder Json and vocab bpe and they do some light processing on it and then they call this encoder object which is the tokenizer now if you'd like to inspect these two files which together constitute their saved tokenizer then you can do that with a piece of code like this um this is where you can download these two files and you can inspect them if you'd like and 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 us to decode very efficiently and basically it took us from the integer to the byes uh for that integer so our vocab is exactly their encoder and then their vocab bpe confusingly is actually are merges so their BP merges which is based on the data inside vocab bpe ends up being equivalent to our merges so uh basically they are saving and loading the two uh 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 now the only thing that um is actually slightly confusing inside what opening ey does here is that in addition to this encoder and a decoder they also have something called a bite encoder and a bite decoder and this is actually unfortunately just kind of a spirous implementation detail and isn't actually deep or interesting in any way so I'm going to skip the discussion of it but what opening ey does here for reasons that I don't fully understand is that not only have they this tokenizer which can

encode and decode but they have a whole separate layer here in addition that is used serially with the tokenizer and so you first do um bite encode and then encode and then you do decode and then bite decode so that's the loop and they are just stacked serial on top of each other and and it's not that interesting so I won't cover it and you can step through it if you'd like otherwise this file if you ignore the bite encoder and the bite decoder will be algorithmically very familiar with you and the meat of it here is the what they call bpe function and you should recognize this Loop here which is very similar to our own y Loop where they're trying to identify the Byram uh a pair that they should be merging next and then here just like we had they have a for Loop trying to merge this pair uh so they will go over all of the sequence and they will merge the pair whenever they find it and they keep repeating that until they run out of possible merges in the in the text so that's the meat of this file and uh there's an encode and a decode function just like we have implemented it 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 BP to organizer train it and then both encode and decode the next topic I would like to turn to is that of special tokens so in addition to tokens that are coming from you know raw bytes and the BP merges we can insert all kinds of tokens that we are going to use to delimit different parts of the data or introduced to create a special structure of the token streams so in uh if you look at this encoder object from open AIS gpd2 right here we mentioned this is very similar to our vocab you'll notice that the length of this is 50257 and as I mentioned it's mapping uh and it's inverted from the mapping of our vocab our vocab goes from integer to string and they go the other way around for no amazing reason um but the thing to note here is that this the mapping table here is 50257 where does that number come from where what are the tokens as I mentioned there are 256 raw bite token tokens and then opena actually did 50,000 merges so those become the other tokens but this would have been 50256 so what is the 57th token and there is basically one special token and that one special token you can see is called end of text so this is a special token and it's the very last token and this token is used to delimit documents ments in the training set so when we're creating the training data we have all these documents and we tokenize them and we get a stream of tokens those tokens only range from Z to 50256 and then in between those documents we put special end of text token and we insert that token in between documents and we are using this as a signal to the language model that the document has ended and what follows is going to be unrelated to the document previously that said the language model has to learn this from data it it needs to learn that this token usually means that it should wipe its sort of memory

of what came before and what came before this token is not actually informative to what comes next but we are expecting the language model to just like learn this but we're giving it the Special sort of the limiter of these documents we can go here to Tech tokenizer and um this the gpt2 tokenizer uh our code that we've been playing with before so we can add here right hello world world how are you and we're getting different tokens but now you can see what if what happens if I put end of text you see how until I finished it these are all different tokens end of text still set different tokens and now when I finish it suddenly we get token 50256 and the reason this works is because this didn't actually go through the bpe merges instead the code that actually outposted tokens has special case instructions for handling special tokens um we did not see these special instructions for handling special tokens in the encoder dopy it's absent there but if you go to Tech token Library which is uh implemented in Rust you will find all kinds of special case handling for these special tokens that you can register uh create adds to the vocabulary and then it looks for them and it uh whenever it sees these special tokens like this it will actually come in and swap in that special token so these things are outside of the typical algorithm of uh B PA en coding so these special tokens are used pervasively uh not just in uh basically base language modeling of predicting the next token in the sequence but especially when it gets to later to the fine tuning stage and all of the chat uh gbt sort of aspects of it uh because we don't just want to Del limit documents we want to delimit entire conversations between an assistant and a user so if I refresh this sck tokenizer page the default example that they have here is using not sort of base model encoders but ftuned model uh sort of tokenizers um so for example using the GPT 3.5 turbo scheme these here are all special tokens I am start I end Etc uh this is short for Imaginary mcore start by the way but you can see here that there's a sort of start and end of every single message and there can be many other other tokens lots of tokens um in use to delimit these conversations and kind of keep track of the flow of the messages here now we can go back to the Tik token library and here when you scroll to the bottom they talk about how you can extend tick token and I can you can create basically you can Fork uh the um CL 100K base tokenizers in gp4 and for example you can extend it by adding more special tokens and these are totally up to you you can come up with any arbitrary tokens and add them with the new ID afterwards and the tikken library will uh correctly swap them out uh when it sees this in the strings now we can also go back to this file which we've looked at previously and I mentioned that the gpt2 in Tik toen open I.P we have the vocabulary we have the pattern for splitting and then here we are registering the single special token in gpd2 which was the end of text token and we saw that it has this ID in GPT 4 when they defy this here you see that the pattern has changed as we've discussed but also the special tokens

have changed in this tokenizer so we of course have the end of text just like in gpd2 but we also see three sorry four additional tokens here Thim prefix middle and suffix what is fim fim is short for fill in the middle and 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 and then there's one additional uh serve token here so that's that encoding as well so it's very common basically to train a language model and then if you'd like uh you can add special tokens now when you add special tokens you of course have to um do some model surgery to the Transformer and all the parameters involved in that Transformer because you are 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 and typically this row would be initialized uh 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 you have to make sure that that projection at the very end into the classifier uh 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 okay so at this point you should have everything you need in order to build your own gp4 tokenizer now in the process of developing this lecture I've done that and I published the code under this repository MBP so MBP looks like this right now as I'm recording but uh the MBP repository will probably change quite a bit because I intend to continue working on it um in addition to the MBP repository I've published the this uh exercise progression that you can follow so if you go to exercise. MD here uh this is sort of me breaking up the task ahead of you into four steps that sort of uh build up to what can be a gp4 tokenizer and so feel free to follow these steps exactly and follow a little bit of the guidance that I've laid out here and anytime you feel stuck just reference the MBP repository here so either the tests could be useful or the MBP repository itself I try to keep the code fairly clean and understandable and so um feel free to reference it whenever um you get stuck uh in addition to that basically once you write it you should be able to reproduce this behavior from Tech token so getting the gb4 tokenizer you can take uh you can encode the string and you should get these tokens and then you can encode and decode the exact same string to recover it and in addition to all that you should be able to implement your own train function uh which Tik token Library does not provide it's it's again only inference code but you could write your own train MBP does it as well and that will allow you to train your own token vocabularies so here are some of the code inside M be mean bpe uh shows the token vocabularies that you might obtain so on the left uh here

we have the GPT 4 merges uh so the first 256 are raw individual bytes and then here I am visualizing the merges that gp4 performed during its training so the very first merge that gp4 did was merge two spaces into a single token for you know two spaces and that is a token 256 and so this is the order in which things merged during gb4 training and this is the merge order that um we obtain in MBP by training a tokenizer and in this case I trained it on a Wikipedia page of Taylor Swift uh not because I'm a Swifty but because that is one of the longest um Wikipedia Pages apparently that's available but she is pretty cool and um what was I going to say yeah so you can compare these two uh vocabularies and so as an example um here GPT for merged I in to become in and we've done the exact same thing on this token 259 here space t becomes space t and that happened for us a little bit later as well so the difference here is again to my understanding only a difference of the training set so as an example because I see a lot of white space I supect that gp4 probably had a lot of python code in its training set I'm not sure uh for the tokenizer and uh here we see much less of that of course in the Wikipedia page so roughly speaking they look the same and they look the same because they're running the same algorithm and when you train your own you're probably going to get something similar depending on what you train it on okay so we are now going to move on from tick token and the way that open AI tokenizes its strings and we're going to discuss one more very commonly used library for working with tokenization inlm and that is sentence piece so sentence piece is very commonly used in language models because unlike Tik token it can do both training and inference and is quite efficient at both it supports a number of algorithms for training uh vocabularies but one of them is the B pair en coding algorithm that we've been looking at so it supports it now sentence piece is used both by llama and mistal series and many other models as well it is on GitHub under Google sentence piece and the big difference with sentence piece and we're going to look at example because this is kind of hard and subtle to explain is that they think different about the order of operations here so in the case of Tik token we first take our code points in the string we encode them using mutf to bytes and then we're merging bytes it's fairly straightforward for sentence piece um it works directly on the level of the code points themselves so so it looks at whatever code points are available in your training set and then it starts merging those code points and um the bpe is running on the level of code points and if you happen to run out of code points so there are maybe some rare uh code points that just don't come up too often and the Rarity is determined by this character coverage hyper parameter then these uh code points will either get mapped to a special unknown token like ank or if you have the bite foldback option turned on then that will take those rare Cod points it will encode them using utf8 and then the individual bytes of that encoding will be translated into tokens and there are these

special bite tokens that basically get added to the vocabulary so it uses BP on on the code points and then it falls back to bytes for rare Cod points um and so that's kind of like difference personally I find the Tik token we significantly cleaner uh but it's kind of like a subtle but pretty major difference between the way they approach tokenization let's work with with a concrete example because otherwise this is kind of hard to um to get your head around so let's work with a concrete example this is how we can import sentence piece and then here we're going to take I think I took like the description of sentence piece and I just created like a little toy data set it really likes to have a file so I created a toy. txt file with this content now what's kind of a little bit crazy about sentence piece is that there's a ton of options and configurations and the reason this is so is because sentence piece has been around I think for a while and it really tries to handle a large diversity of things and um because it's been around I think it has quite a bit of accumulated historical baggage uh as well and so in particular there's like a ton of configuration arguments this is not even all of it you can go to here to see all the training options um and uh there's also quite useful documentation when you look at the raw Proto buff uh that is used to represent the trainer spec and so on um many of these options are irrelevant to us so maybe to point out one example Das Das shrinking Factor uh this shrinking factor is not used in the B pair en coding algorithm so this is just an argument that is irrelevant to us um it applies to a different training algorithm now what I tried to do here is I tried to set up sentence piece in a way that is very very similar as far as I can tell to maybe identical hopefully to the way that llama 2 was strained so the way they trained their own um their own tokenizer and the way I did this was basically you can take the tokenizer model file that meta released and you can um open it using the Proto protuff uh sort of file that you can generate and then you can inspect all the options and I tried to copy over all the options that looked relevant so here we set up the input it's raw text in this file here's going to be the output so it's going to be for talk 400. model and vocab we're saying that we're going to use the BP algorithm and we want to Bap size of 400 then there's a ton of configurations here for um for basically pre-processing and normalization rules as they're called normalization used to be very prevalent I would say before llms in natural language processing so in machine translation and uh text classification and so on you want to normalize and simplify the text and you want to turn it all lowercase and you want to remove all double whites space Etc and 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 you want to keep the raw data as much as possible um in a raw form so you're basically trying to turn off a lot of this if you can the other thing that sentence piece does is that it has this concept of sentences so sentence piece it's back it's kind of like was developed I think early in the days where there

was um an idea that they you're training a tokenizer on a bunch of independent sentences so it has a lot of like how many sentences you're going to train on what is the maximum sentence length um shuffling sentences and so for it sentences are kind of like the individual training examples but again in the context of llms I find that this is like a very spous and weird distinction like sentences are just like don't touch the raw data sentences happen to exist but in raw data sets there are a lot of like inet like what exactly is a sentence what isn't a sentence um and so I think like it's really hard to Define what an actual sentence is if you really like dig into it and there could be different concepts of it in different languages or something like that so why even introduce the concept it it doesn't honestly make sense to me I would just prefer to treat a file as a giant uh stream of bytes it has a lot of treatment around rare word characters and when I say word I mean code points we're going to come back to this in a second and it has a lot of other rules for um basically splitting digits splitting white space and numbers and how you deal with that so these are some kind of like merge rules so I think this is a little bit equivalent to tick token using the regular expression to split up categories there's like kind of equivalence of it if you squint T it in sentence piece where you can also for example split up split up the digits uh and uh so on there's 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 and it hardcodes the UN token the beginning of sentence end of sentence and a pad token um and the UN token must exist for my understanding and then some some things so we can train and when when I press train it's going to create this file talk 400. model and talk 400. wab I can then load the model file and I can inspect the vocabulary off it and so we trained vocab size 400 on this text here and these are the individual pieces the individual tokens that sentence piece will create so in the beginning we see that we have the an token uh with the ID zero then we have the beginning of sequence end of sequence one and two and then we said that the pad ID is negative 1 so we chose not to use it so there's no pad ID here then these are individual bite tokens so here we saw that bite fallback in llama was turned on so it's true so what follows are going to be the 256 bite tokens and these are their IDs and then at the bottom after the bite tokens come the merges and these are the parent nodes in the merges so we're not seeing the children we're just seeing the parents and their ID and then after the merges comes eventually the individual tokens and their IDs and so these are the individual tokens so these are the individual code Point tokens if you will and they come at the end so that is the ordering with which sentence piece sort of like represents its vocabularies it starts with special tokens then the bike tokens then the merge tokens and then the individual codo tokens and all these raw codepoint to tokens are the ones that it encountered in the training set so those individual code points are all the the entire set of code points that

occurred here so those all get put in there 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 uh vocabulary once we have a vocabulary we can encode into IDs and we can um sort of get a list and then here I am also decoding the indiv idual tokens back into little pieces as they call it so let's take a look at what happened here hello space on so these are the token IDs we got back and when we look here uh a few things sort of uh jump to mind number one take a look at these characters the Korean characters of course were not part of the training set so sentence piece 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 un tokens unknown tokens but because bite fall back as true instead sentence piece falls back to bytes and so it takes this it encodes it with utf8 and then it uses these tokens to represent uh those bytes and that's what we are getting sort of here this is the utf8 uh encoding and in this shifted by three uh because of these um special tokens here that have IDs earlier on so that's what happened here now one more thing that um well first before I go on with respect to the bitef back let me remove bite foldback if this is false what's going to happen let's retrain so the first thing that happened is all the bite 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 because we're not taking up space in the wab size uh with all the bytes and now if we encode this we get a zero so this entire string here suddenly there's no bitef back so this is unknown and unknown is an and so this is zero because the an token is token zero and you have to keep in mind that this would feed into your uh 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 so that's why I think llama correctly uh used by fallback true uh because we definitely want to feed these um unknown or rare code points into the model and some uh some manner the next thing I want to show you is the following notice here when we are decoding all the individual tokens you see how spaces uh space here ends up being this um bold underline I'm not 100% sure by the way why sentence piece switches whites space into these bold underscore characters maybe it's for visualization I'm not 100% sure why that happens uh but notice this why do we have an extra space in the front of hello um what where is this coming from well it's coming from this option here um add dummy prefix is true and when you go to the documentation add D whites space 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 tick tokenizer world as uh token by itself has a different ID than space world so we have

this is 1917 but this is 14 Etc so these are two different tokens for the language model and the language model has to learn from data that they are actually kind of like a very similar concept so to the language model in the Tik token World um basically words in the beginning of sentences and words in the middle of sentences actually look completely different um and it has to learned that they are roughly the same so this add dami prefix is trying to fight that a little bit and the way that works is that it basically uh adds a dummy prefix so for as a as a

part of pre-processing it will take the string and it will add a space it will do this and that's done in an effort to make this world and that world the same they will both be space world so that's one other kind of pre-processing option that is turned on and llama 2 also uh uses this option and that's I think everything that I want to say for my preview of sentence piece and how it is different um maybe here what I've done is I just uh put in the Raw protocol buffer representation basically of the tokenizer the too trained so feel free to sort of Step through this and if you would like uh your tokenization to look identical to that of the meta uh llama 2 then you would be copy pasting these settings as I tried to do up above and uh yeah that's I think that's it for this section I think my summary for sentence piece from all of this is number one I think that there's a lot of historical baggage in sentence piece a lot of Concepts that I think are slightly confusing and I think potentially um contain foot guns like this concept of a sentence and it's maximum length and stuff like that um otherwise it is fairly commonly used in the industry um because it is efficient and can do both training and inference uh it has a few quirks like for example un token must exist and the way the bite fallbacks are done and so on I don't find 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 um and just visualizing things and trying to really understand what is happening here because uh the documentation unfortunately is in my opion not 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 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 vocap 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 so this here was uh the file that we built in the previous video and we defined the Transformer model and and let's specifically look at Bap size and where it appears in this file so here we Define the voap size uh at this time it was 65 or something like that extremely small number so this will grow much larger you'll see that Bap size doesn't come up too much in most of these layers the only place that it comes up to is in exactly these two

places here so when we Define the language model there's the token embedding table which is this two-dimensional array where the vocap size is basically the number of rows and uh each vocabulary element each token has a vector that we're going to train using back propagation that Vector is of size and embed which is number of channels in the Transformer and basically as voap size increases this embedding table as I mentioned earlier is going to also grow we're going to be adding rows in addition to that at the end of the Transformer there's this LM head layer which is a linear layer and you'll notice that that layer is used at the very end to produce the logits uh which become the probabilities for the next token in sequence and 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 and 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 do here in this linear layer for this final layer in a Transformer so why can't vocap size be infinite why can't we grow to Infinity well number one your token embedding table is going to grow uh your linear layer is going to grow so we're going to be doing a lot more computation here because this LM head layer will become more computational expensive number two because we have more parameters we could be worried that we are going to be under trining some of these parameters so intuitively if you have a very large vocabulary size say we have a million uh tokens then every one of these tokens is going to come up more and more rarely in the training data because there's a lot more other tokens all over the place and so we're going to be seeing fewer and fewer examples uh for each individual token and you might be worried that basically the vectors associated with every token will be undertrained as a result because they just don't come up too often and they don't participate in the forward backward pass in addition to that as your vocab size grows you're going to start shrinking your sequences a lot right and that's really nice because that means that we're going to be attending to more and more text so that's nice but also you might be worrying that two large of chunks are being squished into single tokens and so the model just doesn't have as much of time to think per sort of um some number of characters in the text or you can think about it that way right so basically we're squishing too much information into a single token and then the forward pass of the Transformer is not enough to actually process that information appropriately and so 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 and it seems like in state-of-the-art architectures today this is usually in the high 10,000 or somewhere around 100,000 today and the next consideration I want to briefly talk about is what if we want to take a pre-trained model and we want to extend the vocap size and this is done fairly commonly

actually so for example when you're doing fine-tuning for cha GPT um a lot more new special tokens get introduced on top of the base model to maintain the metadata and all the structure of conversation objects between a user and an assistant so that takes a lot of special tokens you might also try to throw in more special tokens for example for using the browser or any other tool and so it's very tempting to add a lot of tokens for all kinds of special functionality so if you want to be adding a token that's totally possible Right all we have to do is we have to resize this embedding so we have to add rows we would initialize these uh parameters from scratch to be small random numbers and then we have to extend the weight inside this linear uh so we have to start making dot products um with the associated parameters as well to basically calculate the probabilities for these new tokens so both of these are just a resizing operation it's a very mild model surgery and can be done fairly easily and it's quite common that basically you would freeze the base model you introduce these new parameters and then you only train these new parameters to introduce new tokens into the architecture um and so you can freeze arbitrary parts of it or you can train arbitrary parts of it and that's totally up to you but basically minor surgery required if you'd like to introduce new tokens and finally I'd like to mention that actually there's an entire design space of applications in terms of introducing new tokens into a vocabulary that go Way Beyond just adding special tokens and special new functionality so just to give you a sense of the design space but this could be an entire video just by itself uh this is a paper on learning to compress prompts with what they called uh gist tokens and the rough idea is suppose that you're using language models in a setting that requires very long prompts while these long prompts just slow everything down because you have to encode them and then you have to use them and then you're tending over them and it's just um you know heavy to have very large prompts so instead what they do here in this paper is they introduce new tokens and um imagine basically having a few new tokens you put them in a sequence and then you train the model by distillation so you are keeping the entire model Frozen and you're only training the representations of the new tokens their embeddings and you're optimizing over the new tokens such that the behavior of the language model is identical uh to the model that has a very long prompt that works for you and so it's a compression technique of compressing that very long prompt into those few new gist tokens and so you can train this and then at test time you can discard your old prompt and just swap in those tokens and they sort of like uh stand in for that very long prompt and have an almost identical performance and so this is one um technique and a class of parameter efficient fine-tuning techniques where most of the model is basically fixed and there's no training of the model weights there's no training of Laura or anything like that of new parameters the the parameters that you're training are now just the uh

token embeddings so that's just one example but this could again be like an entire video but just to give you a sense that there's a whole design space here that is potentially worth exploring in the future the next thing I want to briefly address is that I think recently there's a lot of momentum in how you actually could construct Transformers that can simultaneously process not just text as the input modality but a lot of other modalities so be it images videos audio Etc and how do you feed in all these modalities and potentially predict these modalities from a Transformer uh do you have to change the architecture in some fundamental way and 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 the day and pretend it's just text tokens and just do everything else identical in an identical manner so here for example there was a early paper that has nice graphic for how you can take an image and you can chunc at it into integers um and these sometimes uh so these will basically become the tokens of images as an example and uh these tokens can be uh hard tokens where you force them to be integers they can also be soft tokens where you uh sort of don't require uh these to be discrete but you do Force these representations to go through bottlenecks like in Auto encoders uh also in this paper that came out from open a SORA which I think really um uh blew the mind of many people and inspired a lot of people in terms of what's possible they have a Graphic here and they talk briefly about how llms have text tokens Sora has visual patches so again they came up with a way to chunc a videos into basically tokens when they own vocabularies and then you can either process discrete tokens say with autog regressive models or even soft tokens with diffusion models and uh all of that is sort of uh being actively worked on designed on and is beyond the scope of this video but just something I wanted to mention briefly okay now that we have come quite deep into the tokenization algorithm and we understand a lot more about how it works let's loop back around to the beginning of this video and go through some of these bullet points and really see why they happen so first of all why can't my llm spell words very well or do other spell related tasks so fundamentally this is because as we saw these characters are chunked up into tokens and some of these tokens are actually fairly long so as an example I went to the gp4 vocabulary and I looked at uh one of the longer tokens so that default style turns out to be a single individual token so that's a lot of characters for a single token so my suspicion is that there's just too much crammed into this single token and my suspicion was that the model should not be very good at tasks related to spelling of this uh single token so I asked how many letters L are there in the word default style and of course my prompt is intentionally done that way and you see how default style will be a single token so this is what the model sees so my suspicion is that it wouldn't be very good at

this and indeed it is not it doesn't actually know how many L's are in there it thinks there are three and actually there are four if I'm not getting this wrong myself so that didn't go extremely well let's look look at another kind of uh character level task so for example here I asked uh gp4 to reverse the string default style and they tried to use a code interpreter and I stopped it and I said just do it just try it and uh it gave me jumble so it doesn't actually really know how to reverse this string going from right to left uh so it gave a wrong result so again like working with this working hypothesis that maybe this is due to the tokenization I tried a different approach I said okay let's reverse the exact same string but take the following approach step one just print out every single character separated by spaces and then as a step two reverse that list and it again Tred to use a tool but when I stopped it it uh first uh produced all the characters and that was actually correct and then It reversed them and that was correct once it had this so somehow it can't reverse it directly but when you go just first uh you know listing it out in order it can do that somehow and then it can once it's uh broken up this way this becomes all these individual characters and so now this is much easier for it to see these individual tokens and reverse them and print them out so that is kind of interesting so let's continue now why are llms worse at uh non-english langu and I briefly covered this already but basically um 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 um is not sufficiently trained on non-english data and so here for example hello how are you is five tokens and its translation is 15 tokens so this is a three times blow up and so for example anang is uh just hello basically in Korean and that end up being three tokens I'm actually kind of surprised by that because that is a very common phrase there just the typical greeting of like hello and that ends up being three tokens whereas our hello is a single token and so basically everything is a lot more bloated and diffuse and this is I think partly the reason that the model Works worse on other languages uh coming back why is LM bad at simple arithmetic um that has to do with the tokenization of numbers and so um you'll notice that for example addition is very sort of like uh there's an algorithm that is like character level for doing addition so for example here we would first add the ones and then the tens and then the hundreds you have to refer to specific parts of these digits but uh these numbers are represented completely arbitrarily based on whatever happened to merge or not merge during the tokenization process there's an entire blog post about this that I think is quite good integer tokenization is insane and this person basically systematically explores the tokenization of numbers in I believe this is gpt2 and so they notice that for example for the for um four-digit numbers you can take a look at whether it is uh a single token or whether it is two tokens that is a 1 three or a 2 two or a 31 combination and so all the different numbers are all the different

combinations and you can imagine this is all completely arbitrarily so and the model unfortunately sometimes sees uh four um a token for for all four digits sometimes for three sometimes for two sometimes for one and it's in an arbitrary uh Manner and so this is definitely a headwind if you will for the language model and it's kind of incredible that it can kind of do it and deal with it but it's also kind of not ideal and so that's why for example we saw that meta when they train the Llama 2 algorithm and they use sentence piece they make sure to split up all the um all the digits as an example for uh llama 2 and this is partly to improve a simple arithmetic kind of performance and finally why is gpt2 not as good in Python again this is partly a modeling issue on in the architecture and the data set and the strength of the model but it's also partially tokenization because as we saw here with the simple python example the encoding efficiency of the tokenizer for handling spaces in Python is terrible and every single space is an individual token and this dramatically reduces the context length that the model can attend to cross so that's almost like a tokenization bug for gpd2 and that was later fixed with gp4 okay so here's another fun one my llm abruptly halts when it sees the string end of text so here's um here's a very strange Behavior print a string end of text is what I told jt4 and it says could you please specify the string and I'm I'm telling it give me end of text and it seems like there's an issue it's not seeing end of text and then I give it end of text is the string and then here's a string and then it just doesn't print it so obviously something is breaking here with respect to the handling of the special token and I don't actually know what open ey is doing under the hood here and whether they are potentially parsing this as an um as an actual token instead of this just being uh end of text um as like individual sort of pieces of it without the special token handling logic and so it might be that someone when they're calling do encode uh they are passing in the allowed special and they are allowing end of text as a special character in the user prompt but the user prompt of course is is a sort of um attacker controlled text so you would hope that they don't really parse or use special tokens or you know from that kind of input but it appears that there's something definitely going wrong here and um so your knowledge of these special tokens ends up being in a tax surface potentially and so if you'd like to confuse llms then just um try to give them some special tokens and see if you're breaking something by chance okay so this next one is a really fun one uh the trailing whites space issue so if you come to playground and uh we come here to GPT 3.5 turbo instruct so this is not a chat model this is a completion model so think of it more like it's a lot more closer to a base model it does completion it will continue the token sequence so here's a tagline for ice cream shop and we want to continue the sequence and so we can submit and get a bunch of tokens okay no problem but now suppose I do this but instead of pressing submit here I do here's a tagline for ice cream shop

space so I have a space here before I click submit we get a warning your text ends in a trail Ling space which causes worse performance due to how API splits text into tokens so what's happening here it still gave us a uh sort of completion here but let's take a look at what's happening so here's a tagline for an ice cream shop and then what does this look like in the actual actual training data suppose you found the completion in the training document somewhere on the internet and the llm trained on this data so maybe it's something like oh yeah maybe that's the tagline that's a terrible tagline but notice here that when I create o you see that because there's the the space character is 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 are token 8840 that's that's space o so what's What's Happening Here is that when I just have it like this and I let it complete the next token it can sample the space o token but instead if I have this and I add my space then what I'm doing here when I incode this string is I have basically here's a t line for an ice cream uh shop and this space at the very end becomes a token 220 and so we've added token 220 and this token otherwise would be part of the tagline because if there actually is a tagline here so space o is the token and so this is suddenly a of distribution for the model because this space is part of the next token but we're putting it here like this and the model has seen very very little data of actual Space by itself and we're asking it to complete the sequence like add in more tokens but the problem is that we've sort of begun the first token and now it's been split up and now we're out of this distribution and now arbitrary bad things happen and it's just a very rare example for it to see something like that and uh that's why we get the warning so the fundamental issue here is of course that um the llm is on top of these tokens and these tokens are text chunks they're not characters in a way you and I would think of them they are these are the atoms of what the LM is seeing and there's a bunch of weird stuff that comes out of it let's go back to our default cell style I bet you that the model has never in its training set seen default cell sta without Le in there it's always seen this as a single group because uh this is some kind of a function in um I'm guess I don't actually know what this is part of this is some kind of API but I bet you that it's never seen this combination of tokens uh in its training data because or I think it would be extremely rare so I took this and I copy pasted it here and I had I tried to complete from it and the it immediately gave me a big error and it said the model predicted to completion that begins with a stop sequence resulting in no output consider adjusting your prompt or stop sequences so what happened here when I clicked submit is that immediately the model emitted and sort of like end of text token I think or something like that it basically predicted the stop sequence immediately so it had no completion and so this is why I'm getting a warning again because we're off the data distribution and the model is just uh predicting just totally arbitrary

things it's just really confused basically this is uh this is giving it brain damage it's never seen this before it's shocked and it's predicting end of text or something I tried it again here and it in this case it completed it but then for some reason this request May violate our usage policies this was flagged um basically something just like goes wrong and there's something like Jank you can just feel the Jank because the model is like extremely unhappy with just this and it doesn't know how to complete it because it's never occurred in training set in a training set it always appears like this and becomes a single token so these kinds of issues where tokens are either you sort of like complete the first character of the next token or you are sort of you have long tokens that you then have just some of the characters off all of these are kind of like issues with partial tokens is how I would describe it and if you actually dig into the T token repository go to the rust code and search for unstable and you'll see um en code unstable native unstable token tokens and a lot of like special case handling none of this stuff about unstable tokens is documented anywhere but there's a ton of code dealing with unstable tokens and unstable tokens is exactly kind of like what I'm describing here what you would like out of a completion API is something a lot more fancy like if we're putting in default cell sta if we're asking for the next token sequence we're not actually trying to append the next token exactly after this list we're actually trying to append we're trying to consider lots of tokens um that if we were or I guess like we're trying to search over characters that if we retened would be of high probability if that makes sense um so that we can actually add a single individual character uh instead of just like adding the next full token that comes after this partial token list so I this is very tricky to describe and I invite you to maybe like look through this it ends up being extremely gnarly and hairy kind of topic it and it comes from tokenization fundamentally so um 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 and it just okay so this comes from this blog post uh solid gold Magikarp and uh this is um internet famous now for those of us in llms and basically I I would advise you to uh read this block Post in full but basically what this person was doing is this person went to the um token embedding stable and clustered the tokens based on their embedding representation and this person noticed that there's a cluster of tokens that look really strange so there's a cluster here at rot e stream Fame solid gold Magikarp Signet message like really weird tokens in uh basically in this embedding cluster and so what are these tokens and where do they even come from like what is solid gold magikarpet makes no sense and then they found bunch of these tokens and then they notice that actually the plot thickens here because if you ask the model about these tokens like you ask it uh some very benign question like please can you repeat back to me the string sold gold Magikarp

uh then you get a variety of basically totally broken llm Behavior so either you get evasion so I'm sorry I can't hear you or you get a bunch of hallucinations as a response um you can even get back like insults so you ask it uh about streamer bot it uh tells the and the model actually just calls you names uh or it kind of comes up with like weird humor like you're actually breaking the model by asking about these very simple strings like at Roth and sold gold Magikarp so like what the hell is happening and there's a variety of here documented behaviors uh there's a bunch of tokens not just so good Magikarp that have that kind of a behavior and so basically there's a bunch of like trigger words and if you ask the model about these trigger words or you just include them in your prompt the model goes haywire and has all kinds of uh really Strange Behaviors including sort of ones that violate typical safety guidelines uh and the alignment of the model like it's swearing back at you so what is happening here and how can this possibly be true well this again comes down to tokenization so what's happening here is that sold gold Magikarp if you actually dig into it is a Reddit user so there's a u Sol gold Magikarp and probably what happened here even though I I don't know that this has been like really definitively explored but what is thought to have happened 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 redded data potentially where the user solid gold Magikarp was mentioned in the text because solid gold Magikarp was a very common um sort of uh person who would post a lot uh this would be a string that occurs many times in a tokenization data set because it occurs many times in a tokenization data set these tokens would end up getting merged to the single individual token for that single Reddit user sold gold Magikarp so they would have a dedicated token in a vocabulary of was it 50,000 tokens in gpd2 that is devoted to that Reddit user and then what happens is the tokenization data set has those strings but then later when you train the model the language model itself um this data from Reddit was not present and so therefore in the entire training set for the language model sold gold Magikarp never occurs that token never appears in the training set for the actual language model later so this token never gets activated it's initialized at random in the beginning of optimization then you have forward backward passes and updates to the model and this token is just never updated in the embedding table that row Vector never gets sampled it never gets used so it never gets trained and it's completely untrained it's kind of like unallocated memory in a typical binary program written in C or something like that that so it's unallocated memory and then at test time if you evoke this token then you're basically plucking out a row of the embedding table that is completely untrained and that feeds into a Transformer and creates undefined behavior and that's what we're seeing here this completely undefined never before seen in a training behavior and so any of these

kind of like weird tokens would evoke this Behavior because fundamentally the model is um is uh uh out of sample out

of distribution okay and the very last thing I wanted to just briefly mention point out although I think a lot of people are quite aware of this is that different kinds of formats and different representations and different languages and so on might be more or less efficient with GPD tokenizers uh or any tokenizers for any other L for that matter so for example Json is actually really dense in tokens and yaml is a lot more efficient in tokens um so for example this are these are the same in Json and in yaml the Json is 116 and the yaml is 99 so quite a bit of an Improvement and so in the token economy where we are paying uh per token in many ways and you are paying in the context length and you're paying in um dollar amount for uh the cost of processing all this kind of structured data when you have to um so prefer to use theal over Json and in general kind of like the tokenization density is something that you have to um sort of care about and worry about at all times and try to find efficient encoding schemes and spend a lot of time in tick tokenizer and measure the different token efficiencies of different formats and settings and so on okay so that concludes my fairly long video on tokenization I know it's a try I know it's annoying I know it's irritating I personally really dislike the stage what I do have to say at this point is don't brush it off there's a lot of foot guns sharp edges here security issues uh AI safety issues as we saw plugging in unallocated memory into uh language models so um it's worth understanding this stage um that said I will say that eternal glory goes to anyone who can get rid of it uh I showed you one possible paper that tried to uh do that and I think I hope a lot more can follow over time and my final recommendations for the application right now are if you can reuse the GPT 4 tokens and the vocabulary uh in your application then that's something you should consider and just use Tech token because it is very efficient and nice library for inference for bpe I also really like the bite level BP that uh Tik toen and openi uses uh if you for some reason want to train your own vocabulary from scratch um then I would use uh the bpe with sentence piece um oops as I mentioned I'm not a huge fan of sentence piece I don't like its uh bite fallback and I don't like that it's doing BP on unic code code points I think it's uh it also has like a million settings and I think there's a lot of foot gonss here and I think it's really easy to Mis calibrate them and you end up cropping your sentences or something like that uh because of some type of parameter that you don't fully understand so so be very careful with the settings try to copy paste exactly maybe where what meta did or basically spend a lot of time looking at all the hyper parameters and go through the code of sentence piece and make sure that you have this correct um but even if you have all the settings correct I still think that the algorithm is kind of inferior to

what's happening here and maybe the best if you really need to train your vocabulary maybe the best thing is to just wait for M bpe to becomes as efficient as possible and uh that's something that maybe I hope to work on and at some point maybe we can be training basically really what we want is we want tick token but training code and that is the ideal thing that currently does not exist and MBP is um is in implementation of it but currently it's in Python so that's currently what I have to say for uh tokenization there might be an advanced video that has even drier and even more detailed in the future but for now I think we're going to leave things off here and uh I hope that was helpful bye

and uh they increase this contact size from gpt1 of 512 uh to 1024 and GPT 4

two the next okay next I would like us to briefly walk through the code from open AI on the gpt2 encoded

ATP I'm sorry I'm gonna sneeze and then what's Happening Here is this is a spous layer that I will explain in a bit What's Happening Here

is