[SQUEAKING] [RUSTLING] [CLICKING]

RAMA RAMAKRISHNAN: So let's start with a quick review. Last week, we looked at BERT, how BERT was created. And we learned about this technique called masking, which is a kind of self-supervised learning. And the idea of masking was very simple. We asked ourselves the question. We have seen ways in which people can take images and pre-trained models like ResNet on vast body of images. But then for each image, somebody had to go and label them.

So for text, we asked the question, well, what does it mean to label a piece of text when we don't actually have a clearly defined end goal in mind, except the general goal of pre-training things? And then we said oh, well, what we can do is we can actually replace some of the words in every sentence with what we call a mask token. And then we just train the network to recover the blanks, to fill in the blanks.

And this technique, which is one of many ways of doing what's called self-supervised learning, is called masking. And we described how, if you essentially take all of Wikipedia, and for every sentence, you mask it like this, and then train a network to recover, to fill in the blanks, the resulting network becomes really good at doing all kinds of interesting things. And that, in fact, the first such network, or one of the first such networks, was called BERT. And in fact in your homework, you've been looking at BERT and so on and so forth. That's masking.

Now we're going to switch gears and talk about a different kind of self-supervised learning, which is different from masking, which turns out to be weirdly more interesting and powerful. So we are going to look at another technique. And this technique is called next word prediction. So now it is actually, in some sense, a special case of masking, where you're basically saying, take a sentence and, instead of randomly picking a word and making it a blank, you're saying, I'm just going to take the last word and make it a blank.

And then you send the sentence in. And then you have the machine just fill in the blank on the last word, predict the next word. And you don't have to use full sentences for it. You can use parts of sentences for it, sentence fragments as well. So if you take the same sentence as before, the mission of the MIT Sloan School, you can literally divide it into-- well, you can give the and ask it to predict mission. If you can give it the mission and ask it to predict of. You give it the mission, of, ask to predict the. You get the idea.

So every sentence fragment, you can take, and literally just give it the first few and then predict the next one, first few, next one, first few, next one. So this is next word prediction. And so what we're going to do now is we're going to actually take the transformer encoder architecture that we used to build BERT in the last class. And we're going to try to use it to solve next word prediction, to build a model that can do next word prediction.

So this is what we have. So what we're going to do is, if you take the phrase, the cat sat on the mat, so the phrase was, let's say, the cat sat on the mat, so what you might want to do is to say, OK, this is the input. Output, the cat. Then maybe you have the cat. And then the output is sat. The cat sat on, and so on. You get the idea.

And then, finally, we have the cat sat, whoa, whoa, whoa, whoa, the mat. This, basically, what we have, all these inputs and outputs. But we're going to very compactly express it as if we're just coming in through as one sort of data point in one batch. And that's what we're doing here. So what we're going to do is we're going to stack it up like this, where we have the cat sat on the, on the left, meaning everything but the last word.

And then we're going to take that same sentence and just shift it to the left one. So the cat sat on the mat, we cut off the mat, right? And that becomes the input. Then we cut off the first word, and that becomes the output. So when you look at it that way, you can see here, the, you will want the to be used to predict cat, you will want the cat to be used to predict sat, and so on and so forth.

So this is just a little sort of manipulation so that we don't have to have dozens of sentences or sentence examples just for one starting sentence. So if you have something like this, what you can do is you can run it through positional input embeddings, like we have done before with BERT. Then we can run it through a whole bunch of transformers. It's like a transformer stack. Then we get these contextual embeddings.

Then we run them through maybe one or more ReLUs, if you want, because it's always a good idea to stick some ReLUs at the very end. And then we basically attach a softmax to every one of the things that are coming out. And then that softmax is actually going to be a softmax whose range is the entire vocabulary.

For now, let's assume that the vocabulary is just a vocabulary of words, not tokens. We'll get into tokens a bit later on in the class. For now, just assume it's words. And roughly speaking, let's say there are 50,000 words in our vocabulary. So each of these softmaxes-- and this is exactly what we did for BERT, by the way. Each of these softmaxes is like a 50,000 way softmax.

But what we're going to do is, here, when we look at it this way, since we are fundamentally bothered about next word prediction, as you will see later on, we are actually going to ignore all these predictions because who cares? We are only going to look at the last one to figure out, OK, what is the last prediction?

What is it? Because the last prediction is going to be based on everything that came before it here. So this is really the next word that's actually being predicted. All the things before, we don't care so much. And all this will become slightly clearer because we're going to make a couple of passes through it. Yeah?

AUDIENCE: How do we predict the end of a sentence then?

RAMA RAMAKRISHNAN: So the notion of a sentence has disappeared at this point. What we're going to do is, when we look at how we tokenize the input for these kinds of models, we're actually going to take punctuation into account.

AUDIENCE: Oh.

RAMA RAMAKRISHNAN: So we're going to take periods into account, exclamation marks into account, and so on and so forth. And that will answer your question, and we'll come back to that. So this is what we have. So all right, so just to be clear, the embedding that's coming out of the final dense layer is passed through its own softmax with the number of softmax categories equal to the vocab size.

All right, so first of all, so let's say we train models, a model like this, with lots of inputs and outputs. This just looks like BERT. It's not that different, except that there is no notion of a mask. Do you notice any problems with the way this thing has been set up?

AUDIENCE: For some words, like the, you're going to have a lot of potential output pairs that come out of that.

RAMA RAMAKRISHNAN: True, which means that if you have a word like the, the next word--

AUDIENCE: Very hard to predict.

RAMA RAMAKRISHNAN: True. So some words may be hard to predict, depending on the last word of the sentence that was the input. That's what you're getting at, yeah. Other concerns? So I want you-- yeah?

AUDIENCE: Since you're using contextual on this, the output of the first word is going to have access to the second word. And so it's kind of like cheating.

RAMA RAMAKRISHNAN: Bingo. So remember, bingo is a technical term in deep learning, which means great. So if you go to this, as she points out, if you look at the self-attention layer, note, remember, the self-attention layer is the key building block of the transformer block. And so in the self-attention layer, every word, we calculate its contextual embedding by weighting-- weighted averaging of its relationship to all other words in the sentence.

So the last word can see the first word, the first word can see the last word, and so on and so forth. But when we are doing next word prediction, this feels problematic because you're peeking into the future. So let's say that you want to predict the next word. If you look at this architecture, what it can simply do, it can simply copy it from the input because it can see the whole sentence.

So if I tell you, hey, the cat sat on the mat, if I just gave you, the cat sat on the, can you predict the next word for me? You'd be like, yeah, duh, it's mat. The whole thing becomes challenging only if I say, the cat sat on the dash. Now, predict the dash. So to put it another way, let's say that you want to predict-- you're fed in the first two words, and you want to predict this. This is the right answer for the prediction.

The network should only use the first two. However, but because self-attention can see sat, it can see this next word, it will trivially learn to predict the next word to be sat. There is no challenge for it. So this is the key problem. This is the key problem with just using the transformer as is.

AUDIENCE: What's our loss function here?

RAMA RAMAKRISHNAN: The loss function in all these things is actually the same as before, which is that, for every output that's coming out-- so imagine you have just a traditional classification problem, in which you have one output, let's say, dividing-- you're classifying things to 10 categories, like we did with the Fashion MNIST, 10 digits.

So you have 10 outputs, and that goes through a softmax. And then you have 10 probabilities. And there we use cross-entropy. So here, for every one of these things, we use cross-entropy. So we take this output, and there's a cross-entropy just for that, plus a cross-entropy for that, and so on and so forth. So we minimize still cross-entropy, but the sum of all these cross-entropies.

AUDIENCE: And does it get complicated at all by the fact that we have a large vocabulary size now, So lots of output?

RAMA RAMAKRISHNAN: It gets complicated just because there are more things to worry about, compute and so on and so forth. But conceptually, no difference. Whether you have 10,000 or 50,000, it's the same thing. It's just that, instead of classifying an input into one of 10 categories, the inputs themselves are as long as the number of words in your sentence.

So each word that comes into your sentence is being classified in one of 50,000 ways. So essentially, you have as many classification problems as you have number of words in a sentence. But at the end of the day, the loss function is the sum of all those things, or to be more precise, the average of all those things.

Actually, I think I may have a slide about this, which I may have hidden because I wasn't sure if I would have time. Let's unhide it. And I did not agree ahead of time that we're going to set this up like this. All right, so, yeah, so we still use a classic-- cross-entropy loss function. So each word that comes in, so the cross-entropy is actually minus log probability of the right answer.

And you may recall this from earlier in the class. So we just do the same thing for cat sat on the, everything, and then we just take the average, 1 over 7. Boom, that's it. So to go back to this problem, so this is the issue. The issue is that we can't allow words to be predicted knowing the future. They should only know about the past words.

So what do we do? We have to make a change to the transformer to make it work for next word prediction. So what we're going to do is when we are calculating the contextual embedding for a word, remember, the context embedding for a word is going to be a weighted average of all the other word's embeddings. We will simply give zero weight to future words. If you give zero weight to future words, it's almost as if they don't exist. And this will become clear in a second.

So imagine that this is the thing we are going to calculate. These are all-- for every word in the sentence, we are calculating the pairwise attention weight. And you will remember, I went through this with an iPad thing last week. We calculate all the weights. So, for example, to find the-- so all these weights in every row will add up to 1.

And so you take the contextual embeddings of the cat sat on the, multiply them by their respective weights that add up to 1, which is the first row of this table. And that gives you the contextual embedding for the word the, so on and so forth. And since we can't look at the future words, all we do is we go take this table and we just zero everything out in red. We just zero everything here out. And then we renormalize so that the remaining cells, the non-zero dot cells, will still add up to one in each row.

So what that means is that, if you're actually only looking at the, only this thing is going to play a role. For cat, only this thing is going to play a role. So let's give an example. So to calculate-- to predict on, you'll only look at the words for the cat sat. The rest of it will not be considered at all.

Now, the effect of doing all this is that-- by the way, this is called causal self-attention. This tweak is called causal self-attention. It's also called masked self-attention. Just different labels for the same thing. And so what that means is that, when you're looking at the input for the, only the is going to be used to predict cat. When you look, the cat, only these two are going to be used to predict sat and so on, and so on, and so forth.

So this thing here, so all we do is we go into our transformer and we just change each attention head to be a causal attention head. And the way it's actually done under the hood is actually very elegant for computational efficiency purposes, but I won't get into it because it gets a bit involved. But the key idea is replace basic plain vanilla attention with causal attention, a.k.a., masked attention.

When you do that, boom, suddenly, it starts working for an expert prediction. It can't cheat anymore. And when we do that, we get the transformer causal encoder. And, by the way, the word causal here, there's no connection to causality. So it's just a term. So if you look at the original transformer paper, it was created for translation, for machine translation, English to German, those kinds of use cases.

So it had something called an encoder, which we are very familiar with from last week. And then it had something called a decoder. And it was called the encoder-decoder architecture. And we're not going to cover the encoder-decoder architecture because we are not covering machine translation in this class. But I'm mentioning this because this part of the architecture is called a decoder because it uses-- see, here there is a masked attention business going on here. Because it is using this masked attention, it's called decoder.

So the transformer encoder is also referred to sometimes as the transformer decoder. But the word decoder has two meanings. It's a synonym for the causal encoder, like we have seen today. It's also used to refer to sequence-to-sequence translation problems for the second part of its architecture. So you just have to keep it-- it will become clear from context what we are talking about. In this course, of course, there is no confusion because we're not going to be looking at translation. We may say decoder, causal encoder. It's the same thing.

AUDIENCE: So I thought there were some transformers that use bidirectional things. Did you-- is it different from [INAUDIBLE]?

RAMA RAMAKRISHNAN: No, all bidirectional means is that I can see everything. So the encoder we looked at last week, the basic self-attention thing is bidirectional. Basically, all it means is I can look in both directions to see, what are the words there? In causal, you're not using the one in the future, correct? All right, so to summarize where we are, this is what we looked at last week for BERT. And this is the transformer encoder.

And we take the same thing. And instead of multi-head attention, we would do causal multi-head attention. We get the decoder, a.k.a., causal encoder. And we use the left for masked prediction. We use the right for next word prediction. All right, so now, instead of having an encoder, if you have a causal encoder, a TCE here, now we can train models for expert prediction using the same exact approach as before.

We set up the inputs and the outputs, like I described earlier. We run it through a bunch of stacks, a stack of causal encoders, dense, ReLU, softmax, and so on and so forth. Otherwise, the details don't change, but all important changes go into the attention layer and make it masked or causal. Any questions so far? Yeah?

AUDIENCE: This would only apply when we're training the model, not when we're validating or testing the model, right?

RAMA RAMAKRISHNAN: So if you give me a sentence after training, the final prediction is the only thing you care about. And by definition, the final prediction will use everything that came before it. So we're OK. Was that your question?

AUDIENCE: No, I think the fact that we're zeroing out the weights with feature words, I thought it would apply more when we're training the model and we're trying to minimize the loss as opposed to when we're asking ChatGPT what the next word or sentence should be.

RAMA RAMAKRISHNAN: Right. But the point is, when we actually use them, what is the objective? What do we want to do when we actually use them for inference, once we finish training? Our objective is given a particular string. Get me the next word. And to find the next word, you can, in fact, use everything that came before it. And, therefore, without any change to this model, it'll just work for your intended purpose. You don't have to go in there and change it to-- you don't have to unmask it for inference because you don't need to. Yes?

AUDIENCE: I have one question that's regarding when we do the puzzle transformers, we are putting certain weights to 0 for the words which are to be predicted, and then we [INAUDIBLE].

RAMA RAMAKRISHNAN: No, the words that are in the future.

AUDIENCE: Future.

RAMA RAMAKRISHNAN: Yeah.

AUDIENCE: And then we normalize it.

RAMA RAMAKRISHNAN: Correct.

AUDIENCE: And if trained a transformer earlier on all the words, all the words together, so won't be there be a difference in weights between both of these?

RAMA RAMAKRISHNAN: Between the two ways of training? The weights are going to be very different. And they are two different models. BERT is used for certain things. And this kind of model, which is the basis of GPT, is going to be used for other things.

AUDIENCE: We are training it as well like that, by putting [INAUDIBLE] by moving some of the weights to 0.

RAMA RAMAKRISHNAN: Correct. So what I'm talking about here is the-- what we are trying to do here is to say, let's say that we want to do next word prediction as the task, as a self-supervised learning task. And we want to train such a model on a vast amount of text data. Well, we can't just use what we did last week because it's not going to work because of the fact you can see the future. Therefore, we make a tweak.

And then we build this model. Now the question becomes, OK, what can we do with such a model? We have basically trained two different kinds of models, the one that can see everything, BERT, and the one that can't see the future, which is actually GPT. So what can you do with it? And we're going to come to that.

All right, so now once you train such a model, given any input sentence, let's say that the sentence is, it was a dark and. It was a dark and. It goes through all these things. And remember what I said earlier. The fact that it's predicting something after just seeing it, we don't really care. What we're really curious about is, what is the next thing it's going to say?

And the next thing it's going to say is going to be, basically, what's coming out of this softmax. Does it make sense? We don't care about anything that went before it because we already have a half-formed sentence. And we want to just find the next thing here. So we only care about this. These things will come out of the architecture of the model, but we throw them out. We don't even pay any attention to them.

We only look at what's coming out in this one here. And what comes out of that softmax, remember, is a 50,000 way table of probabilities. That's what a softmax is. It's a whole bunch of probabilities that add up to 1. And so it's going to-- and let's say, for example, that you have starting with aardvark all the way to zebra. And these are the probabilities.

So it was a dark and, just for kicks, I put stormy as the most-- highest probability number. But these numbers will add up to 1. We have this table. And then what we do is we choose a token from this table. We get to choose. There's a whole bunch of numbers in this table, and we get to choose a token. The simplest thing one can think of is just choose the word that is the most likely, and we choose the word that's most likely here.

And we're going to have a whole section on how to choose these things coming up. For now, let's go with the simple option. We're going to just choose the one that's most likely, 0.6. And then we attach it to the input. So now the input has become, it was a dark and stormy. We run it through. And, again, we only care about the last one softmax.

We do that, we get another table. And this table turns out-- the table keeps changing because the softmax is different for each time you run it through because the input has changed. So you get a new table, and it turns out the most likely one is night. And then we attached-- so night comes out the other end. And we attached night here, and we keep on going.

We can keep on going maybe till we basically tell the model, OK, generate up to 100 tokens and stop. It might stop after 100. Or it might decide, the model may decide, in fact, that when it sees a punctuation, like a period, or an exclamation mark, or something, that's going to stop. And we have control over this when it stops and how it stops.

But this is sort of the basic process. And you folks are all very used to it because you've all been playing with ChatGPT and the like. But the basic building block is next word prediction, feed it back to the input, next word prediction, keep on doing it. You keep on doing it, then, suddenly, it's writing entire novels for you. Yeah?

AUDIENCE: The longer the initial input is, it's better. It's going to give you a better prediction?

RAMA RAMAKRISHNAN: It depends on your objective. So, fundamentally, you have some task you want the thing to do for you. And that task may-- and you need to give it all the information it can possibly find useful. Yeah, so the long-- the more helpful the input, the better. Maybe that's how I would say it. Yeah?

AUDIENCE: Would this also apply to something like Google Search or-- because they also do next letter prediction too, but with this [INAUDIBLE].

RAMA RAMAKRISHNAN: Yeah, so the Google autocomplete, for example, I don't know if they actually use this kind of model under the hood or not. I just don't know. These things tend to be kept tightly under wraps. If they were to-- if they were using it, my guess is that they-- so I don't know if you folks have seen, recently, over the last few months, they have-- there is a generative AI panel that opens up when you do a Google search.

That panel, I suspect, uses this, but I don't know if the default Google autocomplete actually uses it or not because it's very compute heavy. So I don't know what they do. So, yeah, this is what you do. Other questions on this, on the mechanics of it? Yeah?

AUDIENCE: For our vocabulary list, I'm assuming it's static [INAUDIBLE].

RAMA RAMAKRISHNAN: Yeah, correct. And as you will see here, it's not really a word vocabulary. It's a token vocabulary, but, yes, it is static for a given model.

AUDIENCE: And I guess I'm assuming for Google or any other sort of search engine, that wouldn't necessarily be static. And so when it comes to, I guess, base-- or even [INAUDIBLE] because the model would be different [INAUDIBLE].

RAMA RAMAKRISHNAN: Sort of thinking about what happens to new words and things that are formed? And how does it handle it if the vocabulary is static? There's a very elegant solution that's coming up. All right, so now, in other words, we have learned how to do sequence generation. We already saw that we can do classification with BERT. We can do labeling with BERT, BERT-like models, which are trained on mass prediction.

And for generating sequences, now we know how to do it. We just need to use a transformer causal encoder. Now, these kind of models, sequence generation models trained on text sequences using expert prediction, are called autoregressive language models or causal language models.

And, of course, the GPT family is perhaps the most well-known example of an autoregressive language model, autoregressive because people who have done econometrics and some regression know the notion of autoregression means that you predict something, and then you use sort of the past predictions as inputs into the next time you predict. So this is the notion of autoregression. You predict, you feed the prediction back, get the next prediction, and keep on cycling through. Yes?

AUDIENCE: So when you put it in as inputs into GPT, for example, and it has that-- it shows you the next words as it's come in. Is that an indication of it doing this recalculation and what you described?

RAMA RAMAKRISHNAN: Correct. That's exactly what's going on. In fact, if you use the API, that is the thing called the streaming API, where it will actually stream each token that's coming out through every pass. And you can actually see everything very clearly. But when you actually work with the web interface, and you see the thing almost as if it's typing, like a human, what I've heard from people-- I don't know if this is true. What I've heard from people is that they can actually do it much faster. They slow it down intentionally to give you the feeling that it's actually coming from a human.

So it's like a UX trick to slow it down, to make it feel as if someone is actually typing something on the other end. So when you're interacting with a chatbot, for example, sometimes, you will see it actually typing. Slowly, you can see the bubble and you can see the typing. It's actually intentionally slowed down because you know it's a bot otherwise. So there's a little bit of UX creepiness, maybe, going on. I don't know to what extent this is 100% true and how pervasive it is, but folks who work in the field have told me that this actually is not uncommon.

So that's what's going on here. These are language models. And, of course, GPT-3 is an autoregressive language model. And the reason why we have an L in front of the LLM, because it was trained on lots of data with lots of parameters. At some point, it's not a small language model anymore. It's a large language model. Yeah, so it's LLM. Nothing more momentous than that.

So as it turns out, GPT-3 uses 96 transformer blocks, 96 blocks,. And each block has 96 causal attention heads. And you can see-- you can read the GPT-3 paper. It gives you all the details of the architecture. That is interesting because, GPT-4, they didn't publish the architecture. From GPT-3-- after GPT-3, everything became closed. So we actually don't know what the architecture is, even though there's a lot of speculation on Twitter.

But GPT-3, we know exactly what happened. It's 96 blocks. Each has 96 causal attention heads. And then the data was actually, they scraped $30 billion sentences from a whole bunch of sources web text, Wikipedia, a bunch of book databases. And then they basically just took those 30 billion sentences and just trained it exactly, next word prediction. That's it. Now, when they trained GPT-3, I think it cost them a lot of money because things were not as-- we hadn't figured out how to do it as efficiently as we know now, but it was still pretty amazing.

And I'll talk about what is so special about GPT-3 in just a minute or two. So this is what we have here. And as you folks have seen, the notion of generating text is very powerful because we can obviously generate text. But you can also generate code because code is just text. We can generate documentation for code. We can summarize text. We can answer questions. We can do chatbot. The list goes on.

All the excitement we see around GenAI from the time ChatGPT came out is precisely because this simple idea of text in, text out is just so flexible. It's so versatile. It can handle all sorts of use cases. That's why there's so much excitement. By the way, if you're really curious, I would actually recommend seeing this video, where this guy, Andrej Karpathy, builds GPT from scratch.

It's a fantastic video. If you have even a little bit of curiosity about how these things are actually built, I would strongly recommend checking it out. And there's also a little blog post where this person-- basically, if you know NumPy, you can actually create GPT-3, GPT, using NumPy without using any frameworks and things like that. So I found it super interesting and helpful to understand what exactly is going on, so if you would like to do this.

So now we're going to talk about decoding sampling strategies, which is, I said that when we produce-- when we come up with the softmax for that last token, we have 50,000 choices. What do we pick? As it turns out, to actually get really good performance out of GenAI systems, like ChatGPT, you need to be quite thoughtful about how to decode, how to actually sample from that table. So we'll talk about that for a bit.

So first of all, definition, the process of choosing a token from the probability distribution from the-- coming out of the softmax-- I'm sticking this table right here. This is the softmax. This process of choosing it is called decoding. That's the technical term for it. We get this table, we have to decode, meaning we have to pick something from this table. That's called decoding.

Now, there are two sort of extreme cases of very highly simple ways to do this. The first thing, of course, is just pick the one-- just pick the word with the highest probability. This is called greedy decoding. So in this case, for example, if stormy is 0.6, the highest probability in this whole table, we just pick stormy. So that is the obvious extreme simple case.

The other thing we can do, which is also super simple, is that, because we have a probability table here, we can just reach into the table and sample a word out of it in proportion to its probability, which means that, if you have this table and you're sampling from it, if you sample from it 100 times, 60 times, you probably get stormy because the probability is 0.6.

But some small fraction of the time, you may get strange things, like aardvark, and zebra, and so on and so forth. You're just literally doing random sampling. That's a fine way to do it too. Right. There's nothing wrong with that. So these are both options. So the key thing you need to remember is that-- which one you pick. And there are some variations on it, which we'll get to in a moment.

What you pick, which way to decode your pick, really depends on what your task is, what you're trying to use the system for, the LLM for. So the broad thing to remember is that, if you're working on questions for which the factual accuracy of the response is really important and/or you want the output to be deterministic, meaning every time you ask it a particular question, you really want the same answer back.

You can imagine a customer call support agent, where the two different customers ask the same question, and they get different answers. You don't want that. So you want deterministic outputs. So in those situations, you should use greedy decoding. It's a good starting point because you will get-- you won't get any random stuff because, for any given input sentence, the softmax that comes out of the table is not going to change. It's the same table.

And if you're always picking the highest number in the table, that's not going to change either. So guaranteed determinism. And I've found that for reasoning questions and things where you're asking questions, math questions, reasoning questions, logic questions, you should really sort of keep it as greedy as possible in my experience.

Now, there are other situations where random sampling is actually a better option. If you're doing creative things, write a poem, write a haiku, write a screenplay, things like that, you do want a lot of creativity, in which case, actually, randomness is your friend. You get a lot of different varieties of responses, diversity of responses. All that is really good.

The price you pay for it is that, you lose determinism, the outputs are going to be stochastic. They're going to be random. They're going to vary from the same question. The answer is going to vary again and again. But in many cases, maybe it's OK. You don't care. So that's how, roughly, we think about it. The other one I want to say is that the diversity of response is also important because, if you imagine a chatbot, if you ask questions, if the chatbot always responds in the same stilted robotic fashion, it kind of starts to get annoying.

You want some variation in the output because a human will never give you the same thing back. Though I must say that when I interact with call center agents, I think they're just cutting and pasting from a text library, so it does look kind of robotic. So maybe we are already kind of used to this. But anyway, so those are some of the things to keep in mind. Yeah?

AUDIENCE: If you're using random sampling, do you end up with a better estimation of the uncertainty and probabilities that are more calibrated, in the sense that the table that you end up at the end is the real probability that you observe from the words in your corpus?

RAMA RAMAKRISHNAN: The table doesn't change, regardless of how you sample it. The table is a starting point for sampling. All decoding is about what token from the table you're going to pull out.

AUDIENCE: Oh, so it doesn't impact the loss function?

RAMA RAMAKRISHNAN: No. Yeah, all those things are fixed. You literally get the table, and then you literally can forget how you got the table. And now decoding starts.

AUDIENCE: Is that the reason why ChatGPT would generate a different answer given the same prompt if we run it again and again? Because they are using random sampling?

RAMA RAMAKRISHNAN: Correct. That's exactly why. And we'll see-- I'll do a demo of it very shortly because you can actually manipulate it.

AUDIENCE: If you do the prediction word-by-word, is there a way to make it resilient to mistakes? If you say, the night was dark and aardvark, that can mess up the next word, right?

RAMA RAMAKRISHNAN: It can totally mess it up.

AUDIENCE: So how does it get itself back on track?

RAMA RAMAKRISHNAN: It cannot. And so great question. And we'll look at an example of things going off the rails in just a second. Yeah?

AUDIENCE: Is this how Bing works, where you can slide between being more creative and more accurate?

RAMA RAMAKRISHNAN: Yeah, exactly. So Bing has this creative, balanced, precise something. They're basically under the hood. They're manipulating some of the-- we're going to look at some of those parameters in just a moment. They're just manipulating it for you. But if you use the API, you can manipulate it directly. All right, so here is the basic thing to remember about random sampling.

So our hope is that, for any given sentence, we think that there is probably some set of good answers for the next word and a whole bunch of bad answers, intuitively. So we want the probability of the good stuff. We want-- you can imagine a distribution as going like that. There is the head of the distribution, the first few words in the distribution, if you sort them from high to low probability. And then there is all the long tail of kind of inappropriate-- not inappropriate, irrelevant words.

So our hope is that the model is so good that, for any given input phrase, it basically concentrates the output probability in the softmax, so just a few good words, and sort of, kind of zeroes out everything else. That is the ideal scenario because, in that scenario, if you do random sampling, by definition, you'll pick something from the high quality head of the distribution, and life is good.

Now, we want random sampling to sample from the head and not from the tail. That's the key point. And what do I mean by head and tail? Let's be very clear. So imagine you have-- take the table that we looked at, the softmax table, which went from, what, about aardvark to zebra. And let's say we sort the table based on high to low probabilities.

So maybe what's going to happen is that stormy is going to have a probability of, I don't know, 0.6. And I think, if I remember right, a night had a probability of 0.3. And then there were a whole bunch of other words, all the way to the 50,000th word. From highest low probability, so this is what I'm-- so you can think of this as like a probability distribution

And so, basically, what we are saying here is that this is the head of the distribution, while this long tail is the tail of the distribution. And we want our system to grab something from the head and not from the tail because the head is the stuff that's actually the relevant, useful, good stuff. That's really what we're trying to do here. Does it make sense?

So to come back to this-- and here is the most important point to remember about this slide. While the probability of choosing any individual word in this long tail is pretty small-- for any one word, it's pretty small-- the probability of choosing some word from the tail is high. Some word from the tail is high.

So to go back to this thing here, yeah, so in this particular example, 0.6 plus 0.3, there is a 0.9 probability it's going to be either stormy or night. But there is a 10% probability it's going to be one of these words. And who knows what that word might-- it's going to be. It might be some random nonsense word.

So what that means is then this goes to a point from before. If the LLM happens to sample a token from the tail, which is not good, it won't be able to recover from its mistake. It'll just go off the rails, which is why every word that gets generated, it's really important to get it right because it can't recover very often.

AUDIENCE: Is there a technical way to define the difference between the head and the tail?

RAMA RAMAKRISHNAN: No.

[CHUCKLING]

It's like this common thing people use. And the reason why it's not is because it's so problem dependent as to what the-- basically, you're saying that for any particular problem, I think, depending on the question, the right number of words is probably 20. For the same-- for a different question, maybe it's 40. For a totally different model, for the same question, maybe 10. So because of that variability, we just can't figure it out.

So all right, and I'll show you this, how to do this, in just a moment. So just for kicks, I went in to GPT 3.5. And then I said, students at the MIT Sloan School of Management are? And I said, predict the next word. So it turns out, invited is the most likely next word, followed by given, expected, required, and able. These are the top five words. And the probability is 3%, 2%, pretty small probabilities. But then the words that are below it, the remaining whatever, 50,000 odd words, are even lower.

So here, the most likely word is invited. So what I did is I went in there and said, OK, let me try again now with, students at the MIT Sloan School of Management are invited. And now autocomplete that. Find me the next thing. So it comes back with-- see, now, this is my new prompt, students at the MIT School of Management invited to submit their original white papers to the annual MIT something. It seems reasonable. It doesn't seem bad. It seems reasonable.

Now let's mess it up a bit. So now I go in there, and I noticed that the word masters and the word spending were much lower probability than these top five words. I just mucked around until I found these things. So this is only 0.05%. This is 0.11%. So these are clearly in the tail. They're not the most likely. So I said, what's going to happen if I actually force it to use masters? And then I force it to use spending. This is what you get.

Students at the MIT School of Management are masters of chaos.

[CHUCKLING]

They routinely blow past deadlines, fracture. And then I couldn't take it anymore. I stopped it.

[CHUCKLING]

A single word. And then I said, students, students at MIT School of Management, are spending, which is the other unlikely word, the semester learning life skills-- so far, it looks promising-- through knitting socks.

[LAUGHTER]

I'm not making this stuff up. This is GPT 3.5. So, yes, it will go off the rails. You have to be super careful. And so the way we sort of tame random sampling to make it work for us-- yeah?

AUDIENCE: Do you think that these sentences refers-- the past-- the master of chaos routinely blow past deadlines, you think is something that it was in the training set?

RAMA RAMAKRISHNAN: Yeah, that is-- the thing is, it's basically doing some very rough and approximate pattern matching from all the training data it was trained on. So it doesn't mean, for example, that on the mit.edu website, on the collection of sites, that, actually, there were text saying that, yeah, MIT Sloan students were doing all this crazy stuff. It's probably more like a whole bunch of college, university websites probably had some content like that. Maybe there was a bunch of Reddit people posting stuff like that.

So it's just doing some rough pattern matching. It's basically looking-- the thing is, you have to remember, always, with large language models, what it's trying to give you, it's giving you a response that is not implausible. There is no guarantee of correctness. There is no accuracy, nothing like that. It's giving you a probabilistically plausible response. That's it.

Now, us Sloanies being Sloanies, we look at stuff like this and we get offended. So we are imputing our values onto each generation, but it doesn't know and it doesn't care. So, in fact, when I typed in something like, list all the awards that Professor Ramakrishnan has won, it gave me an amazing list of awards. Apparently, I won this and I won that. I won-- now, none of it is true, to which a student said, not yet.

[LAUGHTER]

So I had the TA make a note of that fine person's name.

[LAUGHTER]

Yeah, so that's what's going on. Yeah?

AUDIENCE: I get the sense, maybe there's--

RAMA RAMAKRISHNAN: Could you use the microphone, please?

AUDIENCE: I get the sense that maybe there's some sort of sliding window, that somehow weighting later words more strongly than earlier words, given how far out-- because they do have the context of students at MIT should have steered it in a certain direction, even with the presence of the word masters. So is there something like that happening?

RAMA RAMAKRISHNAN: No, it is just-- the thing is, think about the training process. And the training process, we gave it sentence fragments. And we asked her to predict the next word. Now, clearly, the more you know about the input that's coming and the longer the input, the more clues you have to figure out what the right next word prediction is going to be.

If I say, the capital of, you'll be like, I don't know. It's got to be a country, I guess, or a state, but I don't know anything more than that. But if I say, the capital of France is, dramatic narrowing of the cone of uncertainty. So that's basically what's going on. And, in fact, there's a very beautiful expression I've heard, which is that what elements do, they call it subtractive sculpting.

So what I mean by that is it's sort of like, when you start, it's like this big block of marble. And then every word chips away at the marble. And then when you're done, it's kind of pretty clear. It's David inside the marble. That's sort of what's going on. All right, so to come back to this, what can we do? We can-- there are three ways in which you can tune random sampling to make it work for you.

The first way and the idea of all these things is that you have some probability distribution. We are now going to sort of manually focus on the head. And then we're going to kill everything else, and only focus on the head and sample from that head, which immediately begs the question, how will you decide what the head is? And that was sort of Elena's question from before.

How will you decide what the head is? So one way we do that is to say, you know what? I know we have 50,000 words in the vocabulary. I don't care. Each time, I'm only going to pick the top K words. K could be 10, 20, 30, 40, 50. That's very problem dependent. I'm going to pick the top 20 words, and I'm going to ignore everything else and only sample from the top 10 or the top 20. That's called top-k sampling.

And so the way it works is that let's say this is your whole distribution. And I just stopped at red instead of going all the way to 50,000. And then you see and you decide, let's say that you want k to be 2. So you just grab the top two words, k equals 2. And then you renormalize the probability so they add up to 1. So 0.6 and 0.2, renormalize, it becomes 0.75 and 0.25.

And now, just imagine that this is the new softmax table that you're sampling from. And you grab a number from-- I'm sorry, a word from here and you're done. That's called top-k sampling, very commonly used. But it has a small shortcoming, which is that it basically assumes that this k that you've come up with, let's say 20, every input sentence, the right number of words in the head is 20, which seems-- obviously, it's not a well-supported assumption. It's just an assumption.

So then the question becomes, can we do better? Because what you really want is you want the words that you pick to have the bulk of the probabilities, as much probability as possible. You don't really care how many words are inserted, as long as, together, they have a lot of probability, which brings us to something called top-p sampling, also called nucleus sampling, where instead of deciding on the number of words we're going to pick every time, we decide, you know what?

We're just going to choose all the words, such that the probability of such words that we have chosen is at least p. Sometimes, it may be just two words. Sometimes, it may be 20 words. We don't care. And then we sample from it. So here, same thing here. Let's say you go with p equals 0.9. So 0.6 plus 0.2, 0.8 plus 0.1. 0.9.

Boom, we have hit 0.9. We stop. And then we grab these three words, and then we renormalize them to get this thing. And then, boom, we sample from it. So this, actually, is even more effective in my opinion because it sort of fluctuates. It doesn't hardcode the number of words you think is important. Was there a question? Yeah.

AUDIENCE: What if, let's say, 0.9 ended up-- if foggy was 0.12, would it only be 0.1 from foggy?

RAMA RAMAKRISHNAN: Yeah, what it does is-- so you give it a point, and what it's going to do is it's going to keep adding words till it just crosses that number. Yeah?

AUDIENCE: I was thinking, can't you just set a threshold for the words that don't pick a word below a certain probability? This stormy, what if stormy was 0.89 and then the other one is just 0.10 [INAUDIBLE] so you pick two words?

RAMA RAMAKRISHNAN: Yeah, you can do that. And, in fact, what you can do is you can always say, I want to pick a word, which is the most likely word. You can do that. But if you say, I want a word-- only consider words whose probabilities are at least something, then, basically, what you're saying is that I'm just going to keep on doing. And then we draw a line here.

But the problem is, you don't know how many words have crept over your threshold. You might, for example, find that-- to go to your example, maybe-- you said 0.9 is the threshold. Maybe there are a whole bunch of-- there was a word at 0.89 that you just missed because it didn't make the threshold. You'll be like oh no, I should have made it 0.89.

So there's no right answer, unfortunately. But these are exactly-- this is exactly the kind of thinking that brought us these kinds of ways to tune these things. Sort of the foundation here is that-- the realization that we cannot sort of a priori decide what the right number of words is. So we have to find heuristics to try to do these things.

So in practice, people try all these methods. In fact, you can do both. You can do-- you can set up so that you can do top-p and top-k at the same time. Basically, you're saying, grab words until you cross the probability or you cross k, whichever is earlier. So those are two methods people use heavily. The third method is called distribution. I'm sorry, temperature.

And the idea of temperature is that, in top-k and top-p, it's sort of-- we have to decide on a number up front, k or p. And then we just draw the line and look at the words that pass the threshold. Temperature is like a softer way to do the same thing. It's a softer way to emphasize the head more than the tail. So I think iPad, all right.

So the idea of temperature is, remember, when we have this, whoops, softmax, so aardvark all the way to zebra, you have all these probabilities. Now, remember, where did we get these probabilities? These probabilities came from a softmax. So what is a softmax? We basically had all these nodes, say 50,000 nodes in some output layer. And these were just numbers. Let's just call them a1 through a 50,000.

And then we ran it through a softmax function. And what did it do? It basically did e raised to a1, e raised to a2, all the way to e raised to a, let's call it n. And then it divided it by the sum of all these things to get the probabilities. So this number became e raised to a1 divided by the sum of all the e raised to a's. So e raised to n divided by e raised to a1 plus e raised to a2 and so on and so forth. So this is how softmax works. I'm just refreshing your memory from a few weeks ago.

Now, what temperature does is that-- let me just write it a little easier. So e raised to a1 plus e raised to a2 plus all the way, n. What it does is it introduces a new parameter here called temperature, which is that we divide everything here by T. And the effect of adding this little knob called temperature here is very interesting.

So let's assume for a second that T is a very, very small number. Assume that T is pretty close to 0, very small number. So if T is close to 0, what's going to happen is that, since it's in the denominator here, all these numbers, all these numbers are going to become really big because T is really small. If a1 happens to be a positive number, it's going to become really big. If a1 is a negative number, it's going to be a really, really small negative number.

Now, in particular, what's going to happen is the biggest of all the a numbers, it was already big. Now it's going to get massive, which means that its probability is going to dominate everything else because you're taking a really big number and doing e raised to that number. So what's going to happen is that-- wait, what? What is this?

So if T is close to 0, the biggest a-- hold on one second. The word corresponding to the biggest a will have a probability of 1 or close to 1. And since all the probabilities have to add up to 0, which means that everything else is going to be 0. So the biggest a will have a probability of 1. Everything else is going to have 0. So reducing temperature close to 0 means that the probability distribution is going to peak at the biggest word, and everything is going to become 0.

So in practice, what that means is that if you look at something like this, if you apply temperature here, what's going to happen is that stormy's thing is going to get something like 0.999 and everything else is going to get wiped out. It's going to get really small, it's going to get even smaller, and so on and so forth.

And so when T is exactly 0, basically, what that means is that this is going to be exactly 9-- 1, and everything is going to just get 0. So when one of them is 1 and everything else is 0, when you do sampling from it, you're just picking the big number, which means that it becomes greedy decoding. So that is the value of having temperature as a knob.

Conversely, if you take temperature T and make it bigger and bigger, as opposed to smaller and smaller, this distribution is going to become flat, meaning all the words are going to have the same probability. So any one of these words becomes equally likely. So T close to 0, the biggest word gets picked. T close to, say-- exceeds 1, goes to 1.52, any word becomes likely. It becomes truly random.

So that is the effect of temperature. And this knob, you can actually tune it. All right, so this is called-- I'm at platform.openai.com. It's called the OpenAI Playground. And in this playground, you can actually put in all the sentences you want, you can choose the model, and then you can-- it will actually tell you what the softmax output is. It's very handy.

So this is where I said-- so here are a few things I want to draw your attention to. The first one is, you see temperature here? The default is 1. If you make it 0, it becomes greedy decoding. But you can make it more than 1 if you want. It will give you all kinds of crazy stuff, as you will see in a second. And then they don't have top-k. They don't have support for top-k, OpenAI, but they do have support for top-p.

You can put p here in this thing, and I'll ignore these things. You can read the documentation to understand those things, but you can actually ask it to show the probabilities. So I'm going to ask it to show all the probabilities. And I'm also going to tell it, don't go nuts. Just give me like a few outputs. Let's just call it 30.

And now I'm going to enter some sentences for us to see what's going on. So let's enter the same sentence as before, students at the MIT Sloan School of Management. Or I think that's what we had. So Submit. So this is what's filling out. Now, you go click on this word, you get all the probabilities. Pretty cool.

So you can see invited, given, expected. These are all some of the things we had. And so what you can do is you can go in there and say, here, clearly-- aking? What is that? That's very weird. So I'm going to nuke it again. I'm just going to check to make sure that I use the same sentence as before. It's very brittle. Students, MIT School of Management are-- are. Oh, I know what it is. GPT 3.5.

So let's try that again. So invited, 3.18. That's what we had? Invited 3.19, 3.9. Close enough. So this is what we have. And now, if you wanted to force it to choose invited here, you just go in there and make the temperature 0. Temperature 0 means it's always going to pick the best one, greedy decoding. So you can hit it again. And it better give you invited. See? It's giving you invited. So that's how we manipulate it using temperature.

You can also ask it-- you can also manipulate top-p. You can do all these things. But so people actually use it very heavily for debugging. And when they're playing with a bunch of data with a model for that particular use case, you just play with it to get a sense for what kinds of probability distributions you see.

And then you can fine-tune it using that knowledge. So, yeah, check this out. Oh, I said that, if the temperature goes above 1 to a higher number, every word in that 50,000 becomes sort of equally likely, which means it's going to produce garbage. So let's actually see garbage production in action.

[CHUCKLING]

So all right, let's just nuke this. And I'm going to take the temperature and max it. I'm going to call it 2, which means that literally anything is possible. Submit. Ladies and gentlemen, I present to you a modern large language model.

[LAUGHTER]

[SIDE CONVERSATIONS]

Isn't it shocking? Because when we work with these language models, we have-- when we see it doing some smart things, we always ascribe some level of interesting abilities, and intelligence, and so on. And then you realize, all I had to go in, go in there and change one parameter, and it's garbage. So you can see the amount of garbage it's showing, just by twiddling one parameter. So you have to be in production use cases-- when you're building applications on top of these large language models, you've got to be very, very careful with these parameters. So pay attention.

All right, so what did I have next? So that brings us to sort of the end of the decoding section. Oh, see, now I'm going to switch gears and talk about tokenization, which is that when-- so far, in all the things we have done, including the homeworks and so on, we looked at this tokenization, the standard process for taking a bunch of text and vectorizing it, which was the STIE process, standardize, tokenize, index, and then encode.

And the standardization, I mentioned earlier, strips out punctuation, lowercases everything, sometimes removes stop words like a and things like that. It also does these things called stemming. But it turns out, if you actually work with something like GPT, you know that it hasn't stripped out punctuation. The punctuation is really good. It uses case, uppercase and lowercase. And, in fact, even better, you can actually make up a word as part of your question, and it'll use that word consistently in the output.

So just for fun, I made up a word. I just did this yesterday or day before. I said, here's a new word and its definition. The word is reldoh, hodler backwards. I said the definition, a student who understands deep learning backwards. Please use this word in a sentence. And here is a sentence it's coming up with. I was a little shocked. During the advanced neural network seminar, it became evident that Jane was a true reldoh, effortlessly explaining even the most complex deep learning concepts in reverse order.

[LAUGHTER]

So it clearly knows how to use anything you make up with. So it has the ability to compose things from scratch as opposed to just looking up stuff. So where is the thing coming from? That's the question. And the answer is this very beautiful thing called byte pair encoding, which we'll look at next. So all right, so here, when we look at this process, the disadvantages are, well, some of the things we have discussed, which is that we want to be able to preserve punctuation, we want to be able to preserve case, we want to be able to handle new words, and so on and so forth.

So sort of the modern models, like BERT and so on, they use different tokenization schemes. They don't actually do the STIE thing. And the GPT family uses byte pair encoding, BPE. BERT uses something called wordpiece. All of these ways of encoding, the fundamental idea is to say, well, you know what?

Whatever language you're working with, why don't we start, first of all, with all the individual characters? Because if you could actually work with individual characters, you can clearly compose any word that comes up. Reldoh is just R-E-L-D-O-H, six tokens, if you're working with characters at the character level. But working only with characters is not great because that means that the model, you're giving it no information about the world.

It has to learn every word from scratch, what the word means and so on and so forth. So it would be nice if we can actually give it words as well, but we don't want to give it infrequent words because infrequent words, by definition, are not worth adding to your vocabulary. They're just going to take up another embedding vector and things like that.

For infrequent words, we'll just compose them. We'll actually construct them on the fly because we can always use characters. So we don't want to put every word in there. We only want to put frequent words. But to give this thing the ability to compose new words and not always have to go to characters, we will give it parts of words. These are called subwords.

So the key idea is that, let's come up with a way to build a vocabulary which has characters, full words that are frequent enough to be worth adding, and subwords, or word fragments that occur frequently enough to be worth adding. So, for example, the word standardize, i-z-e, normalize, standardize, and so on and so forth, ize is going to show up a lot in many places.

So you don't want to have standardized, and normalized, and so on. You just want to have ize. You can just attach it to all kinds of words and make it all work. So that's the basic idea of all these tokenization schemes. And BPE is one such way to figure out how to actually construct this vocabulary from a training corpus.

And, by the way, when I say characters, this will include not just uppercase, lowercase, alphabets, and numbers. It will also include punctuation, so that all these things just become atomic units. All right, so what we're going-- the way BPE works is that we're going to start with each character as a token. And I'll talk about the rest of the thing on the page in just a moment. Don't worry about it. We'll start with each character as a token.

So let's say that your training corpus is just a single sentence, the cat sat on the mat. And even though GPT does not actually do any lowercasing, it'll just actually use-- T-H-E uppercase is different than t-h-e lowercase. Just for simplicity, I'm just going to standardize it here. So it just becomes the cat sat on the mat.

And then I'm going to write it in this form, where I basically put a comma after every word. And then I put a little underscore to show the space between the words. I'm going to write it in this format, and it'll become clear why I'm writing it in just a second. Now, my starting vocabulary is just all the individual letters in the training corpus. So the starting is just whatever, all these letters. That's it.

And this is a starting point. And now what we do-- and this is the key step-- we merge tokens that most frequently occur right next to each other. So if two characters or two tokens are occurring right next to each other a lot, let's just merge them because they seem to be occurring a lot together. May as well merge them.

And so here, for example, I've listed the frequency of the adjacent tokens. So, for example, if you look at t, h, t, h shows up right after each other here. It also shows up here. So, therefore, it shows up twice. Now, h, e, again, is showing up here. It's also showing up here. So that also shows up twice. c, a on the other hand, is only showing up here. It's not showing up anywhere else. So it shows up once.

a, t shows up three times, in mat, sat, and in cat, and so on and so forth. You get the idea. So you're just looking at pairwise adjacent tokens. And you pick the most frequent one that's showing up, which in this case happens to be a, t. And then you take a and t and you merge them. So it becomes at.

So when you do that, when you merge them and then you add that new token that you have just literally created to your vocabulary list, and then you update the corpus to reflect the merge you just did, so now the corpus becomes, the cat sat on the mat. But in this case, there is no a and t separately. There is just the at combo token here. Are we good with this step so far?

Take the most frequent things and merge them. It's a way to compress the data. In fact, the algorithm came from someone trying to figure out a way to compress data. Think of it this way. Suppose I tell you, I want you to compress a message I'm going to send to you. And then you look at all the tasks, messages you've had to deal with. And it turns out you're finding that certain characters are occurring next to each other all the time.

Maybe, just for argument, let's say ABC shows up ridiculously often in the messaging. And then you'd be like, if it's always showing up all the time together, why treat it as three things? Let me just call it one thing. ABC, that's it. You send a single token called ABC every time you need to send ABC, not A, B, C. That's the basic idea.

So here, if you come here, that's what we have. And then what we do is now we do, again, this calculation of adjacency tokens on this updated corpus. And you can see here, t, h shows up once, t, h shows up here twice. So you get two. H, e shows up twice. Everything else shows up once. And, yeah, when many things are showing up with equal frequency, just pick one randomly from this.

So we pick up t,h and we merge that, which means that we add t, h to our vocabulary. And once we do that we update the corpus. And now we have-- t, h is now one thing fused together, along with the previous thing, a, t, that had been fused together. That is the corpus after the second merge. And then we do the same thing. We find the frequency of adjacent tokens. So it turns out t, h and e are showing up twice. Everything else is showing up once. So we take t, h, e, merge it to get the. Boom, the. And now we have the cat sat on the mat.

So this process continues till we reach a predefined limit for our vocabulary. Now, as it turns out, when they built GPT-2 and GPT-- let me just see. I think I did some digging around on this thing. Yeah, so GPT-2 and 3, they set the vocabulary size to be roughly 50,000. So it basically kept on doing this till it hit our limit of 50,000, then it stopped. GPT-4 on the other hand, actually, went-- goes all the way to 100,000 vocabulary size.

So this is BPE in action. And so what's going to happen is once you finish this thing, and you have vocabulary, and you have all these things that you have merged, when a new piece of text comes in, the merges-- remember, here, we merged a, t to get at. This t, h became this and so on. When a new piece of text arrives, the tokenization will apply the merges in the exact same order.

So if the new text that comes in is the rat, it's first going to apply the a, t to at to become-- fuse this here and then going to fuse t, h to get this. And then it's going to fuse t, h and e to get that. And the final list of tokens that goes in to your model is going to be the token for the, the token for space, and the token for r, and the token for at.

So let's see this in action. OpenAI has its own thing, but I found this site to be really good. So let's tokenize hands-on deep learning. So you can see here, look at this. So H, uppercase H, is its own token. It's token number 39. And its own token. Dash is its own token. On is its own token.

And then space deep is its token, and space learning is its token. Note one thing, suppose you had said-- let's just say you just had deep, deep learning. Deep has a different token than space deep. What they have realized is that most words are actually going to show up after the space, after a space, much more likely.

So having a space attached to the beginning of the word, it sort of saves you a lot of compute and so on and so forth because they will, in fact, arrive almost all the time with the space before it. That's the way they have attached the space to the word itself. And note that deep learning, deep and deep-- actually, let's call it this way. So deep and deep are different. There is deep. There is-- so, clearly, it's taking case into account. Then I put an exclamation here. Boom, that.

And so, ultimately, what goes in when you have a phrase like, sat on the mat-- so the cat sat on the mat. And you can see here, uppercase The. And then let's just do another thing here. So uppercase The with a space is 383. Lowercase the is 262. And then that's distinct from just the without any space that's a different thing. So these are all the tokens.

Now, let's try something. Let's try Jane. So Jane is one token, which is great. And is another token. Let's see, Rama. Oh, darn. My name wasn't worthy enough to be its own token. But strangely enough, I was very surprised by this. So if I put Rama in lowercase, it's its own token.

[LAUGHTER]

I have no idea what they were scraping, which websites. And if I put Jane here, now j has become its token with space and ane has become different. So the tokenization is a very interesting thing, and it works in very interesting ways, but that's the basic idea, what's going on under the hood. I would encourage you to check out your names to see if it's actually been tokenized. So all right, I'm done. Thanks, folks. I'll see you on Wednesday.

[APPLAUSE]