[RUSTLING]

[SQUEAKING]

[CLICKING]

**RAMA RAMAKRISHNAN:** OK, so today we start the natural language processing sequence. And so just to give you a quick idea, we're going to start with what's called vectorization, and then the bag of words model. And then we'll spend a fair amount of time on a Colab. And then on Wednesday, we talk about these things called embeddings, which you will come to appreciate over the next couple of weeks, form the core atomic unit of all modern natural language processing, and for that matter, vision processing as well.

And then the following week, we'll do transformers, two lectures on transformers. We'll get into the theory, and then we'll get into a bunch of applications. And then lectures 9 and 10 will be all about LLMs. So it's going to be a lot of fun. This is one of my favorite segments of the class. Of course, truth be told, every segment of the class is my favorite, so don't judge me. All right, so let's get going.

So why natural language processing? In some sense, the things I have on the slide here are obvious, but I think it's actually worth reminding ourselves of how important text is for everything we do. Obviously, human knowledge is mostly encoded as text. The internet is mostly text. At least this was true until the advent of TikTok and YouTube. And human communication is mostly text. And cultural production-- movies, books, arts and so on, so much of it is so text heavy.

And so, in some sense, text forms not just a big chunk of all the media that's out there, but it also happens to be the way in which we think and communicate and so on and so forth. So its primacy is, in my opinion, unparalleled in how we think about the world. And so the tantalizing possibility is that, imagine if we had an AI system which could just read and quote unquote, "understand" all this text.

And so you can imagine such a system reading all of PubMed, reading all the medical literature, and then coming back and saying, for this particular disease, this particular protein is actually the malfunctioning protein. And for that, that small molecule is going to dock into the protein and cure the disease. And you didn't know this-- it came back and told you that. Wouldn't it be unbelievable? So my feeling is that such things are going to happen. It's just that it's not going to happen soon enough for my lifetime. But perhaps it'll happen in yours.

All right, so let's continue. So NLP is in action all around us. According to Google, apparently Google autocomplete, which uses a fair bit of NLP, saves 200 years of typing time, apparently, every day. I wasn't very impressed with this number, frankly, because billions of searches are being done every day, and I'm like, only 200 years? So anyway, but I think the more important point is that it made mobile possible.

If you didn't have autocomplete, people would not be typing and pecking on their keyboards. It's going to be much worse. You would have had a hugely dampening effect on e-commerce, for instance. So this humble little autocomplete has incredible, incredible impact on the world economy.

And the other thing which I heard about, I'm not sure if it's 100% true, but it's an interesting example, apparently the very first iPhone keyboard that came out, the soft keyboard, not the hard keyboard, they had some very basic word continuation prediction going on. And so when you start typing t and h, obviously it's going to guess the e is going to come next. So that part is old, old news. Nothing new there.

But apparently, the e letter in the keyboard will become slightly bigger. So when your finger goes towards it, it has a better shot of actually connecting with it. So these kinds of things are used to change the UI in real time in a whole bunch of applications, and you just don't even realize it.

And of course, we all know about LLMs at this point. So I asked you to write a limerick about the beauty and power of deep learning yesterday. And it says, "In a world where data flows like a stream, deep learning is more than a dream. It sifts through the noise with an elegant poise, unveiling insights that gleam. Cool, right? All right, so let's get back to work.

So NLP has extraordinary potential for making products and services much, much smarter. And what I want to point out here is that even if you focus on this very, very simple formalism, a bunch of text comes in, a bunch of text goes out. That's it. If you take that very simple text in, text out formalism, this little humble little thing has just an enormous, enormous range of applicability.

So, obviously, you can send a bunch of text in and ask it to classify it, sentiment routed for customer support. You can try to figure out the intent of what the person is asking in search. You can filter it. You can content filter to make sure there's no toxic, abusive stuff going on. I mean, the possibilities for just text classification are numerous. But that's a use case we're all kind of familiar with. So no surprise there.

Now, text extraction we may be less familiar with here. And the idea is that you can actually look at a lot of unstructured textual data and extract all sorts of interesting entities from it. Hedge funds use it very heavily. They will extract all sorts of company information from news articles. And then obviously doctor's notes. There are a whole bunch of NLP startups that will take the doctor-patient conversation, transcribe it, and then extract disease codes, diagnosis codes, medication codes, and things like that.

So the possibilities for this are enormous. Of course, text summarization, and we all have been doing it thanks to ChatGPT. Take text in, and any kind of summary that comes out of the text, it's just text out. And then text generation, of course. We can take text and do marketing, copy sales, emails, market summaries, so on and so forth, including, troublingly for educators, college application essays.

Code generation is a more subtle example of text out because code is just text. So text in, text out also covers text in, code out and question answering. So you can take a bunch of text, you can take a whole bunch of documents, you can add a bit of text to it, which is your question-- and this whole thing at the end of the day is just text-- send it in, and then you can use it to answer questions and therefore create chatbots for all sorts of interesting applications.

And if you look at this example, call centers, that is where a lot of money is being spent right now to build these call center chatbots for text in, text out question answering. And so just if you drill into this, if you imagine taking all the call center transcripts and their internal product documentation, service documentation, FAQs, et cetera, stick it in, you can start to answer these kinds of questions.

OK, yesterday, what are the top reasons why customers were upset with us? What interventions made by the agent actually worked? What did not work? What characterizes the best agents from the rest? How should we grade this particular agent's interaction with this particular customer? How should we change the call center script? How should we coach the agent in real time? Every one of these applications is amenable to this very humble text in, text out model.

And of course, everybody knows this potential because of the advent of large language models. By the way, Google released something called Google Gemini 1.5 Pro a couple of days ago, and it's incredible. It's incredible. And anyway, we'll get back to that later. But the point is that the kind of potential we have is just amazing, even for text in, text out.

And as you would imagine--

**AUDIENCE:** This is all, like, though we are calling it language, this is all primarily English.

**RAMA RAMAKRISHNAN:** There are lots of multilingual models as well. There are multilingual models. By that, I mean models which are specialized to other languages, non-English languages, and models which are truly multilingual, like polyglot models as well. And both of them are available right now. And many, many modern LLMs are actually trained from the get-go to be multilingual in a bunch of what are called high-resource languages, languages which are spoken by lots of people.

But actually, it's funny you should ask that question, because this Google Gemini model that I just described, they actually-- so there is a language called Kalimpong, which is spoken by 200 people in the world. And so a researcher had created one book, which is sort of like a grammar manual for Kalimpong because there are no other written works in that language.

And so what they did is they took a whole bunch of English dialogue and this book, fed it into a Google Gemini Pro 1.5, and it translated it into Kalimpong at human level proficiency. It had never seen it before. So that's an example of this. Yes. So the question text here is all the things you want to translate from English to Kalimpong. The documents here is just one document, singular, the grammar book, the manual. And then what comes out is a translation.

So these models, even when they're not explicitly trained on a different language, if you give them enough of grammar manuals and stuff like that, they may do a pretty decent job from the get-go with no training. It's kind of a shocker. Two years ago, people would be like, that's impossible.

All right, so back to this. And as you folks may already know, and maybe you're, in fact, participating in this gold rush already, lots of people are creating lots of really cool companies to take some of these ideas and actually create really interesting products and services out of them. So if you're not doing it, and if you've been thinking about entrepreneurial stuff, here's a word of advice-- take the plunge.

Dismissed. I'm just kidding. And as you can imagine, enterprise vendors are rushing to add NLP to all their products. Salesforce-- Einstein now has Einstein GPT, Microsoft has Copilot. I mean, the list goes on. Everybody's scrambling and really trying hard to infuse some GPT magic into whatever they're doing. Some of it is real, a lot of it is not.

OK, so let's go to the arc of NLP progress. How did we get to this kind of crazy times that we live in? So if you look at natural language processing, basically the efforts to take language and try to analyze language and do predictions with language and so on and so forth, the first phase of it was just hand-crafted rules based on linguistics.

So these were all linguists who would really understand the grammar of a language, and then they would use their deep knowledge of linguistics to figure out all these rules by which you can process and analyze natural language text. And then this other thing came along, which was a statistical machine learning approach, which basically said, never mind all that complicated knowledge of linguistics and grammar. Why don't we simply count things? Let's count the number of times these two will co-occur.

Now, let's count that, let's count this. Basically, just count a lot. And let's see if it does, if it does for predicting things, say for classifying text and so on. And shockingly, those methods ended up being really good. They ended up being really good. And in fact, they actually were better than the lovingly hand-curated, linguistically-driven rules, so much so this is a famous quote which says, "Every time I fire a linguist, the performance of the speech recognizer goes up." Obviously made in jest, but there is a kernel of truth to it.

So that's where we were. And then deep learning happened in 2012, roughly. And then we had these things called recurrent neural networks, which are based on deep learning, which actually moved the ball forward. And then in 2017, something called the transformer was invented, 2017. And the transformer replaced everything else across the board. So we're just going to leapfrog directly to transformers in HODL. We will not spend any time on recurrent neural networks.

And that is not to say that they're dead. There's a very interesting work which actually is trying to now revive recurrent neural networks to make it work for these kinds of modern LLM kinds of tasks, but it's still very early days. So for now, we'll just focus on transformers.

So the very high-level view of the problem here is that, like most things in deep learning, it's basically fancy regression. There is some variable, x, that comes in. It goes through this very complicated function along with this w, which is the weights, and then out pops an output. That's just the view that you've always had. And so in this case, x happens to be text. y-hat can be text. It could be labels. It could be numbers. It could be anything else. The w is the weights, and the function is a deep neural network.

At this point, when you look at this slide, it should be blindingly obvious. So now, the key question here is, how do you actually represent x? That's the key question. For pictures, for images, we saw that we just took the pixel values, which were light intensity numbers between 0 and 255, and you could just use that directly. But when a sentence comes in, like, "I love deep learning", what do you do? How do you actually represent it?

Because remember, we have to numerical everything that's coming in. So that's the key. And this actually is a very subtle question, very important question, and we'll focus on that today. And then next week, when we look at transformers, we will look at what neural network architecture is best suited to process these text inputs that are coming in. Those are the two big questions we're going to look at.

All right, so processing basics. We're going to follow this very standard process. This is the process by which we take any text that comes in, and we do run it through these four steps. And this process is called text vectorization. And as the name suggests, we are essentially taking text and creating vectors of numbers out of them. Text vectorization. And we'll go through each of these processes one after the other.

So I just find it very useful to just have this acronym S-T-I-E in my head, like "stie", S-T-I-E. Just keep that in mind. It may be helpful. All right, so what we do is, the setup here is that we have a whole bunch of documents. We call it the training corpus. We have a whole bunch of text documents, text data. And as far as we are concerned, you can just imagine it as just lists of long passages. What is a novel? It's just a long passage of text.

So whether it's a novel or a sentence doesn't really matter. We just think of them as a big list of strings, a big list of text. That's our training corpus. And what we do is we take this training corpus and we run it through, and we apply standardization and tokenization, which I will describe, to this entire training corpus upfront. So we first do this.

And standardization is basically-- the default for most applications tends to be this, which is we first strip capitalization and make everything lowercase, and then we remove punctuation and accents and so on and so forth. That's the first thing we do. I'll talk about why we do it in just a moment.

But the mechanics of it are, we do this first, then we look at words like a, the, it, and so on and so forth, basically filler words, which we need to actually make complete sentences, but they may not have any value in predicting things. So we remove them, and they are called stop words. And then finally, we take words which are very similar, which have sort of the same kind of stem or root, and then we just map it to a common representation, like ate, eaten, eating, eaten, all these things will just become, let's say, eats.

And we do that sometimes. So this, we almost always do. This, we often do. And this, we do it sometimes. Now, why do we do any of these things?

**AUDIENCE:** I think we want to try to recognize the essential thing with the word, whether it's eaten or eat, but the essential thing is the eat part. So we want to try to abstract from it the more essential thing.

**RAMA RAMAKRISHNAN:** Right, so why do we need to abstract? You're absolutely correct. We're trying to abstract. Why is there a benefit to doing this abstraction? How about somebody from this side of the room? Oh, yes.

**AUDIENCE:** [INAUDIBLE]

**RAMA RAMAKRISHNAN:** Why is it a good idea to reduce the size of the library?

**AUDIENCE:** To reduce the computation.

**RAMA RAMAKRISHNAN:** Because of the amount of computation needed. So, that is part of the answer. There's another part to the answer, which is? All right, let's swing to the right.

**AUDIENCE:** Because it facilitates comparison between different sets if you are able to standardize it.

**RAMA RAMAKRISHNAN:** OK, so I will go with that. But I think the key thing to realize here is that you want the model-- much like when we talk about computer vision, we said, look, if it's a vertical line, I want to be able to detect it wherever it happens. I don't want the model to think that the vertical line on the left side is different from the vertical line on the right side and then later realize they are the same thing, because you would have wasted valuable capacity learning things which actually happen to be the same because you didn't know it was the same.

So here, if you, for example, take a word and lowercase it, clearly the case of it, whether it's uppercase or lowercase, most of the time, it's not going to matter for anything you want to predict. So you're essentially telling the model, the lowercase version, the uppercase version, they are not different. They're actually the same. And the easiest way to tell the model they are the same is just make everything lowercase. That is the key idea.

And similarly, if you look at stop words, the reason is that these stop words may not help you predict anything. Whether a word a, and, the showed up in a movie review probably does not affect the sentiment of the review. Therefore, let's remove it. So that's a slightly different reason. Stemming is the same reason as the first one, which is that all these words kind of mean the same thing. We don't have to be super precise about it, and so let's just collapse them onto the same thing.

Now, these are all the standard things we do. There are important exceptions to all these things. And we'll come back to the exceptions a bit later. But that is the standard thing we do. Make sense? All right.

So if you look at something like this sentence here, "Hola! What do you picture when you think of travel to Mexico", bah, bah, boom, and then you can see here, this is the standardized version, everything has become lowercase. The H has become small h. The punctuation has disappeared. That's part of standardization. And you can see here that Mexico m has become small, sipping has become sips, think has become thinks, and so on and so forth. So that's an example of standardization at work.

The next thing we do is something very important, and it's called tokenization. So what we do typically is that, OK, now we have standardized everything. We have a bunch of words. We need to now split them into what are called tokens. So the most common default is to just think of a word as a token. We just split on the whitespace. You take each string, and wherever there is whitespace, meaning actual spaces or carriage returns and things like that, boom, you just split on them, and you just create words out of it.

So, for instance, if you have this standardized sentence here, you just split it after every word, and you get this thing. So each of these is now a token. Now, this has some disadvantages. What are some disadvantages of just splitting on the space between words? Yeah.

**AUDIENCE:** I think we lose any context because we look at each word separately. So we don't have any passwords or what's happened [INAUDIBLE.]

**RAMA RAMAKRISHNAN:** Right. So, for example, "the cat sat on the mat" and "the mat sat on the cat" will have the same set. Yeah, so you lose the order. What are some other issues with it?

**AUDIENCE:** For words that should have chained together, like Puerto Vallarta, you lose the fact that that's one name because you separated it.

**RAMA RAMAKRISHNAN:** Correct, exactly. So there are compound words, like father-in-law, for instance. That's one problem. Another problem is that lots of non-English languages, they actually don't have this notion of a space between words. It actually runs one after the other, and the native speakers know from context how to chunk it and break it. So, well, what do we do then? Because you basically will have one word for the whole passage, one token.

The other problem is that there are languages-- German is perhaps the most notable one-- in which you have very long words. I saw a word, which I think I might have it in a slide somewhere, that's, like, this long, which means you realize that something amazing is happening, but the rest of the world hasn't woken up to it yet. It's that feeling. There's a word for that. Amazing, right?

Anyway, so yeah, some words, in Japanese, for example, there's a word called komorebi. Do people know the meaning of the word komorebi? It means the transient beauty of sunlight going through fall foliage. There's a word for that. How cool is that? Anyway, sorry. I love that word. So, back to this.

So we have this thing here. So there are reasons for which splitting on the space between words is not going to work. So what happens is that modern large language models-- so what we have described so far, despite its shortcomings, is actually really good for lots of NLP use cases. If you want to classify text, it's good enough, for instance. But if you want to generate text, like LLMs do, it's not going to work. It's not going to work.

Because you know when you ask ChatGPT a question, it comes back with perfect punctuation. Clearly, punctuation was not stripped. It comes back with a particular upper and lower case. Clearly, that wasn't stripped. You can actually make up new words and ask it to use the new word. It'll use it. Therefore, it's not like it can only recognize a finite set. So there's a very clever scheme called byte-pair encoding, which is invented to do all those things. And I have slides at the end, and if we have time, we'll talk about it.

For now, let's continue this thing. So when this is done for every sentence or every passage in a training data set, we now have a list of distinct tokens. We have a list of distinct tokens. In this simple case, it happens to be all the distinct words that we have seen. That's called the vocabulary. That's called the vocabulary. So now, we move to the third and fourth stages.

In these stages, the indexing and encoding stage, we only work with the vocabulary. And so what we do is, the first thing, the indexing, we assign a unique integer to each distinct token in the vocabulary. So, for instance, let's say that you took a whole bunch of English literature as your training corpus, and you ran it through. Basically, you come up with English dictionary.

So it will have maybe starting with a all the way to zebra, a whole bunch of words. And so I'm just putting 50,000 here because it turns out the GPT family uses something called 50,000 tokens, so I'm just using 50,000. It's not the actual number of words in the English language. It's much more than that. So let's say that we give a number 1 through 50,000, and then we actually also introduce a special token called UNK, which stands for unknown. And we'll come back to this later. And we give unknown the integer 0.

So this is what we mean by indexing. Take the word, the tokens you have identified. Just map it to an integer. That's the indexing step. Then, what we do is we assign a vector to every one of these integers. And that is the encoding step. We assign a vector to each integer. So you have a bunch of distinct words. Under each word, we put an integer on it. And then we take that integer and map it to a vector. Yeah.

Can you use the microphone, please?

**AUDIENCE:** Can you please explain what unknown means?

**RAMA RAMAKRISHNAN:** Yeah. So I'll come back to that. For now, just assume that we have a token called unknown. And the way we are going to use it will become apparent in a few minutes.

**AUDIENCE:** Does it mean there's a base to it, there's a letter or something?

**RAMA RAMAKRISHNAN:** It's a placeholder for something else, which I'll describe shortly. So that's what we have. So let's say that we want to assign a vector to each integer in our vocabulary. And let's assume that we have-- OK, let's say we have 50,000 possible integers because we have 50,000 possible words, and we want to assign a vector so that if you take the vector two different words, they should look different.

Clearly, that's the whole point of mapping from integer to vector. They better be different. What is the simplest way to come up with a vector for each of these tokens?

**AUDIENCE:** It's the same as the index.

**RAMA RAMAKRISHNAN:** Sorry?

**AUDIENCE:** It's the same as the index. It's just a vector one by one by one [INAUDIBLE]

**RAMA RAMAKRISHNAN:** So a vector of zeros and ones, or?

**AUDIENCE:** Just a vector with one dimension.

**RAMA RAMAKRISHNAN:** Oh, I see. Got it. Well, it's creative, but it's a little cheating, right? Because you're essentially putting a square bracket around the number and saying it's a vector. Good try.

**AUDIENCE:** You could try one-hot encoding.

**RAMA RAMAKRISHNAN:** Right. You can try one-hot encoding. So, remember, the list of distinct tokens you have, you can just think of them as the distinct levels of a categorical variable. And you could just use one-hot encoding for it. So what you can do is the simple thing is to do one-hot encoding. And the way it is going to work is that if you have, let's say, 50,000 possible values, the vector is going to be 50,000 long.

It's going to have 0's everywhere except in the index value of whatever the token is. So, for instance, since we said UNK is going to be the first number, 0, it has a 1 here. The zeroth index position has a 1. Everything is 0. a happens to be the second one, so it happens to be 1 in the second position, zero. You get the idea.

So this is zero-hot encoding. We can do this zero-hot encoding, one-hot encoding. And so the dimension of this encoding vector, how long it is, it's basically the number of distinct tokens that you have seen in the training corpus plus one for this UNK thing, which we'll get to. So that is the dimensional-encoding vector, which this is called the vocabulary size. This is called the vocabulary size.

So, at this point, we have created a vocabulary from the training data, training corpus. Every distinct token vocabulary has been assigned a one-hot vector, and we are done with basic preprocessing. So all the text that has come in, every token has been mapped to one potentially very long one-hot vector. Any questions on the mechanics of this before we continue on?

Now, let's see if, when you get a new input sentence in, a new sentence freshly arriving, and we want to feed it into a deep neural network, how will this process actually apply to that new sentence that's coming in? So let's assume that we have completed our S-T-I-E on the training corpus, and it turns out we found only 99 distinct tokens, 99 distinct words. And then we add this UNK thing to it, so we got 100.

So this is our vocabulary. It starts with UNK, a, and then goes all the way to zebra. But there are only 100 of them in total. And just to be very clear, we didn't bother to do things like stemming and stop-word removal and stuff like that, which is why you have words like "the" showing up in this list.

So let's say this input string arrives, "the cat sat on the mat", and then we run it through S-T-I-E. So "the cat sat on the mat" goes through this thing, boom, boom, boom, boom, boom. Then, the output is going to be a table with a bunch of rows and a bunch of columns. Any guesses how many rows and how many columns?

Please, raise your hands. I'll call on you. Yeah, please use the microphone. Go for it.

**AUDIENCE:** I would guess 100 rows and 6 columns.

**RAMA RAMAKRISHNAN:** All right. We'll take a look. 100 and 6 as well as 6 and 100 are both correct. So the way I've done it, it is 6 and 100, and that's exactly right. So the idea is that this is your vocabulary. So the word "the cat sat on the mat", once you change the case of it, it becomes like this. So the "the" happens to be a one-hot vector with a 1 where there is a "the" and 0 everywhere else. I'm not showing all the zeros because it will get too cluttered.

Similarly, cat has a 1 where the cat position is and 0 everywhere else, and so on and so forth. Does that make sense? So the phrase, "the cat sat on the mat" came in as just, whatever, six words, and then it became this 600-entry table.

Now, what is the best way to feed this table to a deep neural network? What can we do? It's not a vector, it's a table. If it's a vector, we know what to do. We just feed it in. We'll just maybe send it to some hidden layer and declare victory at that point. Yeah.

**AUDIENCE:** You need to flatten it.

**RAMA RAMAKRISHNAN:** You would like to flatten it? And how might you do it? Flattening is a reasonable answer, by the way.

**AUDIENCE:** I think you just have to take each column, take the first one, go into each word and just kinda, like--

**RAMA RAMAKRISHNAN:** Yeah, so basically you can take all the first columns and then take the second column and attach it under the first column and so on and so forth. So we can certainly do that. And that's very akin to how we work with images. But there is one downside to that. What is that downside? Yes.

**AUDIENCE:** It's pretty long. I wonder if instead you could for the first word, it's 1, for the second word, it's 2, and then you maintain the order, but you still keep it just as one row.

**RAMA RAMAKRISHNAN:** One row. So we'll come back to what we do about this. But what you're pointing out is it could be very long. Because if each word is a 50,000-long one-hot vector, with just six words, it becomes a 300,000-long vector. Can you imagine taking a 300,000-long vector and sending it into a 100-unit hidden layer? 300,000 times 100 parameters? Too much. Can't learn anything. So that's one issue.

The other issue is that different length texts that are coming in will have different-sized inputs. So here, "the cats are on the mat" has 6 times 50,000. But maybe the cat sat on the mat and the rat ran over to the cat. It becomes even longer. We can't handle variable-sized inputs. The inputs all have to be mapped to the same length. That's another problem.

**AUDIENCE:** So maybe you can sum the columns, basically, and count how many times each word appears since you're using the spatial relationship.

**RAMA RAMAKRISHNAN:** Yes. So both you and [MUTED] are on the same sort of trajectory, which is that we need to somehow take this table and make it into a vector. And there are many ways, like what you folks are describing, to make it into a vector. And it turns out-- this is all the things that we've been discussing so far, the varying length issue and so on.

So, what we can do is we can aggregate all these things. If you just add them up-- this is what you described, I believe-- it's called sum encoding. And if, instead of adding, you just order them, meaning if you look at the column and say, is there any 1 in this column? If there's any 1, I'll put a stick a 1. Otherwise it's 0. It's called multi-hot encoding.

So if you look at this thing, if you literally just go column by column and count everything, OK, there's a 1 here, 1 here. Oh, wait, there are two 2s here, so you put a 2. That's called count encoding. Multi-hot encoding just looks for any 1s and it puts 1. Makes sense? So by the way, there are many ways to take these tables and make them into vectors. These two happen to be very commonly used, and they kind of make common sense.

So this aggregation approach that we just described is called the Bag of Words model. The Bag of Words model. And the reason is that, first of all, this bag that we have has words. Either it counts whether a word exists or not, or it counts how many times the word has appeared, multi-hot versus sum encoding, count encoding. But more importantly, and this goes back to your observation, is that we have lost the order of the words now.

Whether the phrase came in was "the cat sat on the mat" or "the mat sat on the cat", the count encoding and the multi-hot encoding are exactly the same. There's no difference because we're just looking for the presence or absence of words. That's it. We don't care in which order they appear. That's a huge limitation. But shockingly, for many applications, it doesn't matter. It's good enough. So it's called the Bag of Words model.

So this is called the Bag of Words model. Now, does it have any shortcomings? I already talked about the first shortcoming, which is that it loses sequentiality, the order. We lost this order information. We lose the meaning inherent in the order of the words. What are some other issues with it?

**AUDIENCE:** [INAUDIBLE].

**RAMA RAMAKRISHNAN:** What do you mean by that?

**AUDIENCE:** For example, you have six different words [INAUDIBLE].

**RAMA RAMAKRISHNAN:** Right, so there are lots of zeros, not that many ones. So it's a very sparse amount of information, but maybe you're carrying around a lot of information to make it all work. Now, there are some tricks, CS, Computer Science, tricks to handle sparsity in some clever ways, but it is certainly an issue.

Now, the other issue is that, let's say the vocabulary is very long. Each input sentence, whether it's the collected works of William Shakespeare or the phrase "I love you", will have the same length input. We'll have the same length input. Because ultimately, every incoming thing gets mapped into one vector. That feels a little suboptimal. Clearly, the collected works of literature have a lot more stuff going on in them.

So that's the problem. In particular, very, very small things that come in, you'll be spending a lot of compute on those long vectors and processing them. Now, you can mitigate some of this by choosing only the most frequent words. I think the English language, I read somewhere, has roughly 500,000 words or so. But turns out the top 50,000 most frequent words were responsible for just about everything you're going to see, ever.

And the other 50,000 are what's called the long-tail. They almost never happen. You never see them. So you can be very pragmatic and say, I'm not going to take every little word that I see in my vocabulary. I'm going to only take the most frequent words. I'm just going to ignore the rest. I'm just going to ignore the rest.

But if you ignore the rest-- let's say there is one word-- let's take some Shakespeare word, Hamlet. Let's assume that you ignore the word Hamlet from your training corpus. You just delete it because it's not one of the top, most frequent things you have seen. And then somebody sends you a text saying, "Hamlet was a bad prince. Analyze the sentiment of the sentence", well, when you see Hamlet, what is your system going to do?

It's going to look at the Hamlet and say, I can't see it in my vocabulary anywhere. And if it can't see it in the vocabulary, what is the only thing it can do? Replace it with UNK. So that's where UNK comes into the picture. So whenever it can't see something in the vocabulary in a new input, it just replaces it with UNK, which means that if you had ignored Romeo and Juliet and Hamlet in the training corpus, all of them are going to be replaced by the same UNK, which means that we can't distinguish between them anymore.

**AUDIENCE:** So is this where hallucination comes into play here, where it doesn't recognize [INAUDIBLE].

**RAMA RAMAKRISHNAN:** Ah, interesting question. Is this where hallucination comes up? Actually, as it turns out, no. As we will see when we talk about LLMs later, LLMs actually will not have this UNK problem because they use a different tokenization scheme which can handle anything you throw at it, including new stuff you just made up. So we'll come back to that.

So that's what we have. And so what we're going to do is, despite its shortcomings, Bag of Words is actually a really good default for many NLP tasks. And in the spirit of, "do the simple stuff first and do complicated things only if the simple stuff doesn't work", we'll use the Bag of Words model right now. So we'll switch to a Colab and see how it's done.

So here, the application we're going to work with is kind of a fun application. We're going to try to predict the genre of songs. It's a nice classification use case. So we want to take some arbitrary song and then classify it into either hip hop, rock, or pop. And so, for instance, this is the kind of lyrics you're going to see.

And as you will see in this data set, just a quick word of caution the data set does have lyrics which may not be safe for work, as it were. So I'm not going to be exploring the lyrics in the Colab, but I just want you to be aware of it. But it's just some data set that we downloaded from somewhere. It's got all these lyrics.

So we're going to try to classify each verse that we see into one of three things, hip hop, rock or pop. It's a multi-class classification problem. Actually, what is the simplest neural network-based classifier we can build for this problem? So what is the simplest neural network we can build for this problem? So remember, what is the input? The input is going to be a bunch of song lyrics. It's going to be a really long song for all you know. And we're going to use the Bag of Words model.

And let's assume for a moment that we will use multi-hot encoding. We'll create a vocabulary from the song, we'll take all the songs, we'll process them, run it through S-T-I-E, we'll do multi-hot encoding, which means that every song that comes in will be a vector that's how long? It will be as long as the? Correct, as the vocabulary size.

So maybe what comes in is this phrase, since it's supposed to be songs, I'll say something which is probably common to 90% of songs, "I love you". That goes in, it goes into our S-T-I-E process. And then this S-T-I-E process gives us a vector, which is x1, x2, all the way to xv, where v stands for the size of the vocabulary. So that's our input layer all the way.

So, knowing what we know now about deep learning, what can we do next?

**AUDIENCE:** Maybe I'm getting ahead, but wouldn't the classifier just be-- like, the baseline would be classify it as the most common genre?

**RAMA RAMAKRISHNAN:** That is the baseline, correct. And we'll come to the baseline a bit later. But here, I'm saying suppose you wanted to build a neural network model for this. How would you set it up?

**AUDIENCE:** Do you think about the layers that you want?

**RAMA RAMAKRISHNAN:** Right. And what is the simplest thing you can do with a neural network? How many layers?

**AUDIENCE:** No additional layers.

**RAMA RAMAKRISHNAN:** Well, then it becomes problematic with a singlular neural network because it could just be logistical regression.

**AUDIENCE:** One hidden layer.

**RAMA RAMAKRISHNAN:** Yes, thank you. I'm being a little squishy about this because there are some people who would be like, well, even if there's no hidden layers, if you're using ReLUs and this and that and sigma, maybe it's a neural network, and I don't want to get into that, "how many angels in the tip of a pin" argument. So yeah, we need one hidden layer. In this course, we need at least one hidden layer for it to qualify as a neural network. OK, so let's have a hidden layer.

And we'll have a bunch of ReLUs, as usual. Bunch of ReLUs. And I'll ignore all the arrows between them. It's kind of a pain. And then we come to the output layer. And what should the output layer be? How many nodes do we need in the output layer? Three, hip hop, rock, or pop. And then that layer is called what? What activation function? Softmax, perfect. Love it. Love this class.

All right, three things, rock, hip hop, and pop. And this is a softmax right there. And then it's going to give us three probabilities that add up to 1 because of the softmax. So that's our basic network. Perfect. Yeah.

**AUDIENCE:** Why do you need those probabilities at the end if you just want to identify the most likely genre? The softmax just gives you a way to add them all up at once. Why do you need a softmax? Why don't you just take the max value and say it's that?

**RAMA RAMAKRISHNAN:** Oh, interesting question. Why can't we just produce three numbers and grab the maximum number? So, it turns out finding the maximum of a bunch of numbers, that function is not very friendly for differentiation. And ultimately, you want to take this output and run it through a loss function like cross-entropy and then be able to run backprop on it. And so, fundamentally, backpropagation is just differentiation, and it requires everything inside of it to have well-behaved gradients.

And so this little max function is actually not well-behaved, which is why we have a soft version of it, softmax, which makes it easy to differentiate. So I can tell you more about it offline, but that's the quick synopsis. So a lot of tricks you will see in the neural network literature are ways to avoid the problem of having certain-- like, the obvious choice of function will not be well-behaved for differentiation. That's why you need to go through all these other mechanisms.

Much like we couldn't just say accuracy, why don't we just maximize accuracy instead of doing this cross-entropy business? Same reason. All right. So let's come back here. So that's what we created on the thing, "cat sat on the mat" vocabulary thing and so on. And I was playing around with it earlier, and so I found that eight ReLU neurons were pretty good to get the job done. So I'm just going to go with eight ReLU neurons in the hidden layer.

So I think that brings us to the Colab. Yeah, so let's switch to Colab. All right. So that's what we have here. There's a little bit of verbiage here, which just describes what I just talked about. So we'll do the usual things and upload everything, import everything we want, TensorFlow and Keras, and the Holy Trinity of NumPy, pandas, and Matplotlib, set the random seed, as usual, at 42.

There's our S-T-I-E framework here. And the nice thing is that all four of these things, S-T-I-E, are beautifully implemented in Keras as a single, simple layer called the text vectorization layer, which is nice. So we have the text vectorization array right here. And so in our first example, what we'll do is we will use a default standardization, which will just remove punctuation, convert to lowercase. We'll use the default tokenization, which just means split on the space between words. And then we will set the output to multi-hot.

All the things we talked about, Keras will just do it for you automatically. And so output mode, multi-hot, standardize this, split whitespace, and boom, you run the text vectorization thing. And once you do it, Keras creates this textualization layer with these settings, and it's now ready to swing into action.

So what does swing into action actually mean? Well, now you need to actually feed it a training corpus so that it can do all the things it's supposed to do and create the vocabulary for you. And that thing is called the adapt method. So we create a tiny training corpus for us. This is our data set. This is a bunch of words from some of these lyrics. And then what we'll do is we'll take this layer that we just defined here that we have set up here, and then we will ask this layer to actually create the vocabulary using this adapt command.

It will index the vocabulary, and it's done. And once it does it, you can actually ask it for the vocabulary-- this is vocabulary-- using the get_vocabulary command. And so, first of all, how long is the vocab? 17. 17 words, 17 tokens. What are they? You can see here. And you can see, these are all the words. And you can see it has stuck an UNK in the very beginning. It's sort of the default.

By the way, just a little programming tip, if you don't have a ton of programming experience, if you want to print these Python objects, like lists and so on, in a pretty way, one trick that often works is just stick it into a data frame and then print it. Usually, it'll print it in a much better way. So you can see it like that.

So you can see here, UNK, arrays, blah, blah, blah, blah, blah. And you can see integer zeros assigned the UNK token. By the way, how come it picked the word "arrays" as the second entry? Why not something like "an" or why not "a"? How come "a" is not chosen as a second entry? Why did it pick "arrays", do you think?

**AUDIENCE:** Maybe it tried the words that are most influential on the meaning of the sentence to be on the top and less influential.

**RAMA RAMAKRISHNAN:** But at this point, it doesn't know what we're going to use it for, so it has no way to know what word is useful because we haven't told it how we're going to use it. But you're kind of on the right track. So what Keras does is it will find all these tokens, and then it'll actually just sort them by frequency.

So the most frequent, as it turns out, in those four sentences we gave it, happened to be the word "arrays". That's why "arrays" is showing up on top. And you can actually confirm this by going to the little data set. And you can see here "arrays" shows up here and once up here twice. And that's why it came up on top. So that's what we have. And now that we have populated this, we can run any sentence through it easily. Yeah.

**AUDIENCE:** Does it matter that it's on the top?

**RAMA RAMAKRISHNAN:** It doesn't matter. It doesn't matter. The reason why it's helpful later on is because suppose you tell Keras, hey, don't take every word you see here, give me only the most frequent 100 words. I don't want any more than that. It can easily do that. That's the reason. Yeah.

**AUDIENCE:** We could do it after [INAUDIBLE].

**RAMA RAMAKRISHNAN:** This is just the vocabulary. So basically, you give it all these phrases-- it happened to be just four phrases in our example-- and then it finds all the distinct words, and it does all that stuff. And then it has created a vocabulary. At this point, the training corpus you fed it is forgotten. And the only thing that survived this processing is just the vocabulary. That's it. And now, we have to start applying it to any kind of text we want to use it for.

So here, when you come back here, this is what we have. And so what you can do is you can take any sentence and you can just run it through the layer to make sure that it actually is doing the right thing for you. So we'll take this sentence. We will then run it through the text vectorization layer by just passing that sentence into it, and then we can just print it.

So now, it's giving you a tensor. This is a multi-hot encoded tensor with all these ones and zeros. So note that this tensor is 17 units long, which is a good check because our vocabulary is 17 long, so it better match that. Now, recall that the UNK token is at the first location. It's at index 0. And it says that this encoded sentence does have an UNK word. So why is that? What is this UNK word? Anyone can guess?

Well, it turns out to be the word still, I think. Yeah, still is not in our vocabulary, because if the four sentences, which is our training corpus used to build the vocabulary, they had a lot write and rewrite, but there was no still in it anywhere. That's why there is an UNK for it. We can just double-check that by asking Python, is it vocabulary? No, it is not.

OK, now, in the spirit of making small changes to the code to understand what's going on, which is a very useful tip for folks who don't have a ton of programming knowledge, let's say that you send the phrase Sloan, HODL, and DMD. I think you will agree with me that none of these words is in the training corpus. So what is the multi-hot encoded vector for this phrase, Sloan, HODL, DMD?

Three? It's not count encoding. It's multi-hot encoding. It's going to be 1, 0, 0. So you can see here, in this case, remember the vocabulary is 17. So each of these words is going to be a one followed by 16 zeros. And then it's going to multi-hot encode them, which means the three ones in the column just become a one. So you still have only this one.

All right, good. So now now, let's actually get to the data set we have, this 90,000 songs. And it's in this little thing here. We have grabbed the data and cleaned it up, cleaned it up meaning formatting-wise, not content-wise. And then we stuck it in this data frame. And we already have divided it into train, test, and validation for your benefit, so you don't have to worry about it.

Turns out we have almost 49,000 songs in the training set, 16,000 songs in the validation set, and roughly 22,000 in the test set. It's a lot of songs. It's a lot. It's a big data set. So let's just look at the first few. So, "Oh, girl, I can't get ready", "we met on rainy evening", "paralysis through analysis". OK, that, I can relate to as a data science person, but anyway.

By the way, these things are very useful for exploration of any data frames that you might have. It's a Colab feature, just check it out. So anyway, that's the first few rows. Let's look at the last few rows.

"You never listen to me" is pop. "Bimmer, Benz" is hip hop. Yeah, of course. [INAUDIBLE] OK, now to go back to the question of, OK, what could be a good baseline model? We need to understand the proportion of these three classes of songs. So we will do a quick check. It turns out rock is 55%. So if you had to just guess something just naively, you would just guess everything to be rock, and you'd be right 55% of the time.

So now, by the way, the target variable, which tells you whether which of these three genres it is, is actually a dummy variable. So need to, one, hot-encode that. So we'll just turn that this way using the pandas get_dummies function. And when we do that, this is y_train, which contains the dependent variable. And you can see that is one hot-encoded now.

010, 010, 001, and so on and so forth. That's it. So I think the first, I forget it, rock, hip hop, pop or whatever, it's in some order. We'll get to that later. So it's one hot-encoded as well. So that is as far as the data downloading and setup is concerned. Any questions? Yeah.

**AUDIENCE:** This kind of goes back to the transfer learning concept, but do you always want to build your corpus based off of the vocabulary of your training data? Or could you have a pre-compiled, like, somebody already made like a list of the 50,000 most common words?

**RAMA RAMAKRISHNAN:** That's a really good question. Unfortunately, I'm going to punt on it for the moment, because with modern large language models, a number of these NLP tasks for which you had to roll your own and build your own thing, can now be very easily done using large language models without even any further training. Now, the price you pay for it is that you have to use a large language model, which means you have to pay somebody an API call and things like that. And there are other issues with it.

But we will talk a lot about transfer learning for text when we come to it a little later in the NLP sequence. So if I forget, please bring it up again. Yeah.

**AUDIENCE:** Quick clarification on the encoding vector. I noticed it was floats, not ints. If it gets incredibly long, wouldn't that eat into compute time? Is there a reason why it's floats?

**RAMA RAMAKRISHNAN:** Yeah, so question is that when I showed you that tensor, it's written as a continuous number, a floating point number. But we know these are zeros and ones. So why do we have to waste compute capacity by telling the computer that it's all big continuous numbers when it's just a 0 or 1? There are ways to optimize that, but these problems are so small we just don't worry about it.

But when we come to something called parameter-efficient fine-tuning, lecture maybe 10-ish, we will actually exploit that particular fact to make things faster. So that's what we have. So we will do the Bag of Words model. By the way, there's a whole bunch of stuff here that just repeats what I've been telling you in the lecture. So feel free to read it again, but we can ignore it for the moment.

And now, there is a new thing we are doing here. So we are basically saying, look, instead of taking every word you see in these 49,000 songs in the training corpus, it's going to be too many words, just pick the 5,000 most frequent words. And that's what this max_tokens stands for. And so we tell it, all right, do this thing, max_tokens 5,000 and still do multi-hot.

And we are not explicitly saying the standardization and all that stuff because the defaults are what we're going with. Yeah.

**AUDIENCE:** This is for making it more efficient? This is, like, don't waste your time on these thousands of words. Use them more. Just focus on that to make it more efficient.

**RAMA RAMAKRISHNAN:** It makes it more efficient, but there is a related and important point, which is that, fundamentally, the number of tokens you allow this layer to have dictates the size of your vocabulary. And the size of your vocabulary dictates the size of the vector that you feed in. So shorter vectors are better than longer vectors. That's the efficiency point.

The other point is that the longer the input vector, the more the number of parameters the network has to learn because the first layer itself is the size of the input roughly times the size of the hidden layer. So this thing becomes 10 times as long, you have 10 times as many parameters to learn. And given a finite amount of data, the more parameters you have, the worse it's going to do when you actually start using it in the real world.

It's going to overfit heavily. That's why you need to be very careful. Yeah.

**AUDIENCE:** So have you downloaded the data set? Are you still using the vocabulary of the 17 words, or did you broaden it?

**RAMA RAMAKRISHNAN:** No, no, that was just for fun. I'm going to actually build the vocabulary now. It's coming. Yeah, good question. So, let's do that. So first, I define this layer. OK, I just defined it. Now, we actually build the vocabulary by essentially telling it to adapt the layer using essentially the full all basically 49,000 songs in the training data set. That's a long list of songs.

As far as Keras is concerned, you're just looking for a list of strings. So you just give it the list of strings. Instead of four, we're giving it 49,000. The same philosophy applies, so we run it. It's obviously going to take a few seconds to do that because it's 49,000 songs. Oh, it's about five seconds. All right, let's look at the most common 20. We get the vocabulary from our layer.

See, once you adapt the layer and it's built a vocabulary, the layer has sort of been populated with all this information, so you can query it. So you can get the vocab. Top 20 words-- UNK; "the", most frequent word, no surprise; "you", "I", blah, blah, blah. Let's look at the last few. "Dagger", "cheddar", "terrified". Moving on, and then once we have done that, now we actually can vectorize all the data sets we have using this.

And by vectorize, you mean take every string and create the multi-hot encoded vector from it. Yeah.

**AUDIENCE:** Are we doing S-T-I-E because we're keeping stuff like D-E-A, et cetera?

**RAMA RAMAKRISHNAN:** Yeah, we're not strictly doing S-T-I-E, or to put it differently, typically S has lowercase/uppercase, strip punctuation, stemming, stop word removal. Here, the default in Keras happens to not do stemming, not do stop word removal. So we're just going with the default. Thanks for the clarification.

And in fact, in practice, what I find these days is that don't even bother to stem. Don't even bother to remove the stop words. It's going to work well enough. OK, so now, each phrase is a vector. How long is this vector? Each song is now a vector. How long is that vector? 5,000, correct, because that is the size of the vocabulary. Correct.

It's max tokens long, which is 5,000. So if you actually look at-- oh, wait, wait, wait, wait, wait. I haven't run this thing yet. It's going through 49,000. It's going through another 23,000. Fine. So let's run it. OK, now we can see x-train, which is all the training data you have, is a tensor, is a table, with 48,991 rows. And each row is a 5,000-long vector.

All right, good. Now, we will try this simple neural network that we wrote up in class. And now, at this point, this code should be sort of second-nature. Isn't that cool? It's so easy to write the thing. The power of abstraction. So we take Keras' write input. As usual, input layer. We tell it, what is the size of each thing that's coming in? Well, the size of each thing is a 50 max_tokens long vector. So we tell it the shape is max_tokens, and then we run it through a dense layer with eight ReLUs. OK, I'm hurrying.

So we get the outputs. Then, we string the inputs and the outputs into a model, and then we summarize the model. That's it. So we go here. And this has 40,000 parameters. And you can see here, when you go from the input, the 5,000 times 8, that gives you 40,000 plus the 8 neurons have a bias coming in. That's another eight. So you get 40,008.

And we compile it as usual. We use Adam as usual. And because now the output y variable, the y-train variable, is now itself actually one-hot encoded, 010, 001 depending on pop, rock, and so on and so forth. We don't use sparse categorical cross-entropy. We just use plain, old categorical cross-entropy here. And this was explained in lecture last week, so you can revisit it if it's not familiar.

We again report accuracy. So let's compile it. And we get a model. So we'll just run it for 10 epochs with a batch size of 32. And because we have validation data already supplied to us, we don't have to tell Keras, take the training data and keep 20% of it aside for validation. We can literally tell it what validation to use. That's what we're doing here.

All right, so it's running. It's pretty fast. Any questions so far? Yes. Use the microphone, please.

**AUDIENCE:** How do we decide the max total length? We defined the number of 5,000 here, but we do not know how many words would be there in the entire text.

**RAMA RAMAKRISHNAN:** Yeah, so it's a good question. How do you decide on this maximum vocabulary? What do you typically do in practice is that you actually do it without the max tokens, and then you see how long the vocabulary is, and then you can actually get statistics on how frequently the very infrequent words actually show up. And then you'll typically see a dramatic fall off at some point. And you pick that fall-off point, and then set that to be the max tokens.

Perfect. Let's test it. Accuracy is pretty good, 87% on the training and 73 on the validation. We'll do it on the test set. All right, 72%. So we saw earlier the largest class of the three-way is rock, with around 50%. So the naive model is going to get 50% accuracy, and this little neural network model gets you 72%, which is pretty nice.

So now, let's actually kick it up a notch and make it slightly more capable. So the key thing here is that, as has been observed in class already, when you go with a Bag of Words model, we lose all notion of order. The word order clearly matters, and we're kind of ignoring it. So what we do to get around it is-- this is actually really interesting sentence here.

Let's say this is a movie review. "Kate Winslet's performance as a detective trying to solve a terrible crime in a small Pennsylvania town is anything but disappointing". Tricky, tricky thing, because if you look at the words separately, the word "terrible" and "disappointing", you're going to be, like, negative sentiment.

But then if you actually know that the word terrible refers to the crime, not to the movie or anything but disappointing changes the meaning of the word disappointing, you will see, obviously, it's a positive review. So clearly, the words around the word provide valuable clues as to how to interpret that word. And so what we do is, how can we make our little model a bit more capable of recognizing the context around every word? And the way we do it is something called bigrams.

And for bigrams, what we basically do is instead of just taking each word, we take each word and we further take every pair of adjacent words, and those become our tokens. And because we take two adjacent words, they are called bigrams. You can take three adjacent words, trigrams. You get the idea, n-grams. So that's the idea of bigrams.

And so, for example, if had "the cat sat on the mat", you will have "the", "the cat", "cat", "cat sat", "sat". You get the idea. That's what we have. So let's do a little example. And Keras makes it very easy. You literally tell it n-grams equals 2, bigrams. And now, from this, you immediately should know that n-grams equals 1 is the default. That's why we didn't have to specify it.

So, you run it, and then you do "cat sat on the mat" as your training corpus. And then you get the vocabulary. And you can see here, it has created all these nice bigrams for you. And so that's it. All right, now, what we do is we go back to the songs, and we actually tell Keras to not just take each word, but take all the bigrams as well. And hopefully, it'll do a better job of figuring out what the sentiment is.

And now, because when you say, OK, take the top 5,000 words, that's great for single unigrams, as they are called, but when you have bigrams, you have 5,000 possibilities of the first word, maybe 5,000 for the second word. That's a lot of possibilities, 25 million. Now, most of the 25 million possibilities are going to show up in the data, so you don't need to actually make it much larger, but you should make the vocabulary a bit more than 5,000.

So here, we go with, say, 20,000. Otherwise, it's the same, still multi-hot. So let's run it. And now, we will run this now that the layer has been set up with all the right settings, we'll ask it to create the vocabulary, again, by doing exactly what we did before. Create the vocabulary. [INAUDIBLE] seconds.

Bigrams, trigrams, all of them will get much more compute-intensive. That's why you're seeing this. So, all right, let's look at the first 10 words. The first 10 words are all just single words, and that's not surprising because the single words are going to be the more frequent. And then let's look at the last few. "Your mom", "your God", "you shot", "you hell".

All right, let's just index all the data we have, the training validation test sets, using this vocabulary. Perfect. Now, we come to our second model, where we say the incoming shape is now 20,000 long, because we increased max tokens from 5,000 to 20,000. So each thing is a 20,000-long vector. Otherwise, it's the same. And now, we will use this thing called dropout for the first time, which is a regularization thing that I have referred to earlier that I never really described, and I will describe today if we have time.

But I'll first run through the whole demo. So you can just think of dropout as just another layer you can insert. And it's essentially a great way to prevent overfitting. So I just routinely will use it. And I'll talk more about it. So for now, we have this dropout layer in the middle. It receives the input from the dense layer and then sends it to the output layer. The output layer is unchanged. It's a three-way softmax, same model as before.

And now, all right, I'll come back to dropout. So we'll compile it the same way as before. And then I will just fit it for three epochs. If you're interested after class later on, you can actually try it for more epochs and see if it does better. For now, in the interest of time, we'll just do it for three.

I think 72% was the single-word unigram thing we had.

**AUDIENCE:** If you're rerunning this code with the same number of epochs, do you ever expect the accuracy to change?

**RAMA RAMAKRISHNAN:** If you were to run this code on your machine, you would expect it to be roughly the same, but there are some minute differences due to hardware and device drivers and things like.

**AUDIENCE:** If you reran it on your own machine twice, would you ever expect the accuracy to change?

**RAMA RAMAKRISHNAN:** That's actually a very tricky question, because it depends on what else I have been doing in that notebook. If I start fresh and do nothing but that, typically I get the same numbers, typically. But for some reason, I don't get it exactly right. OK, so we come to this. Let's evaluate our little model.

OK, 75%. So it went from 72 to 75, which is actually a meaningful jump, just by using bigrams. And I ran it only for three epochs. If you run it for 10, maybe it's going to do even better. So that is the beauty of this thing. Now, let's just actually do a little demo. We'll try to predict some lyrics. I'll try "another one bites the dust". It's a rock song. I think that's correct, yes?

OK, folks, your turn now. Somebody, tell me your favorite song.

**AUDIENCE:** "Dancing Queen" from ABBA.

**RAMA RAMAKRISHNAN:** I love ABBA. That's awesome. All right, OK, dancing queen lyrics. Verse one, intro-- I don't like that. Let's just go to something without all this metadata. I'll just take the first page, OK?

Here. Are we good? All right, don't let me down, model. Let's predict. Pop. Just about. Yay!

[APPLAUSE]

All right. So, yeah, that's basically the model. But we have five minutes. I want to get back to-- you can play around and put your own lyrics in. Typically, what happens is that the last two years that I've been doing this particular lecture, I've noticed that the songs are always rock songs for some reason. It's the first time getting a pop song from a group that I actually like, so thank you.

All right, let's go back to dropout. So the idea here in dropout is that you have all these-- the input comes in, it goes through a hidden layer, and so on and so forth. So dropout is a layer. And you put this layer just like you use any other layer. And what dropout does is that it takes all the things that are coming into it from the previous layer and randomly decides to replace that number with a zero. That's it.

It drops that number and replaces it with zero. But it does it randomly. It basically will toss a coin. If the coin comes up heads, zero. If it comes up tails, let it through, pass it through. And the reason why this is very effective is because you can imagine all the neurons in a particular layer. When they overfit to a particular data set, the overfitting happens because the neurons essentially collude with each other.

They collude with each other to actually overfit and predict things in a very accurate way. So you want to break any sort of collusion between the neurons. I'm obviously using a game theoretic way of describing it. But the idea is that any kind of spurious correlations in your data, neurons can pick it up by being correlated themselves. And so the way you avoid the spurious correlation is by dropping neurons randomly.

You just kill the neuron randomly, which means that no neuron can depend on another neuron being available. I know, it's a bit grim, but that's the basic idea of dropout. And apparently, the story goes that the team that invented it, Jeff Hinton, who won the Turing for this stuff, not for the dropout, just for deep learning, I don't know if it's true, but he said that apparently he got the idea when he went to a bank and realized that, very often, the folks who are working in that bank branch that he used to go to kept changing.

They were never the same. The people would be transferring in, transferring out. And he was like, why can't they just leave these people alone? Why does it keep changing? And then he got the insight that maybe a lot of fraud happens because the person working in the branch colludes with the customer. But by changing the staff constantly, you break the risk of fraud happening. And that apparently was the genesis for this idea.

True, apocryphal, I have no idea, but it's sort of a fun story. Yes.

**AUDIENCE:** Instead of random, if we go to the way historical models are built, the concepts of multicollinearity and all of that, would that make it sharper as compared to this?

**RAMA RAMAKRISHNAN:** The problem is that these networks are massive. And for you to take each layer and look at its correlation with some other layer and so on and so forth, first of all, investigating multicollinearity is a problem. The second thing is, OK, what do you do then next? In linear regression, you can do things like principal components analysis to get around it. Here, everything is non-linear. There is no easy way to solve the problem. So we are like, we'll just solve the problem in one shot using dropout. That's the idea.

All right, so I had some material on something called byte-pair encoding, which I will do when we get to LLMs. And I stuck it in the end because I knew that we probably wouldn't have enough time to cover this anyway. And that is a very clever tokenization scheme used by, for example, the GPT family, that allows them to do beautiful punctuation, keep the case intact, and then use words that you just made up and things like that.

So we have one more minute. I'm happy to answer any questions you might have.

**AUDIENCE:** So initially when we are picking the hidden layer, the number of neurons, and [INAUDIBLE] to so far all the materials this has been given to us. But initially, how do you pick it? Is it more of a trial and error type of thing?

**RAMA RAMAKRISHNAN:** It tends to be trial and error. That's, in fact, what I did when I created the Colabs. And you can actually make it a bit more systematic by trying lots of different values. And there is a particular Python package called KerasTuner. So just Google KerasTuner, and it comes with very nice Colabs. And if I have a chance, maybe I'll just record a screen walkthrough of doing that. But that's the very efficient way to do these things.

And it comes under the broad category of something called hyperparameter optimization, where the number of neurons, the activation you use, the learning rate, all those things can all be tried. You can try lots of variations. And Keras is a great way to do it in the context of Keras. Other questions? No? All right, I'll give you 30 seconds back. Thank you. See you tomorrow.