[SQUEAKING] [RUSTLING] [CLICKING]

RAMA RAMAKRISHNAN: We'll continue our journey with natural language processing. We looked at the bag of words model, one-hot embeddings, and so on and so forth. And today, we will talk about embeddings, or to be more precise, standalone embeddings. And then that will tee us up for something called contextual embeddings, which is where the transformer really comes into play.

All right. So let's get going. So FR, we have encoded input text with one-hot vector. So to just to refresh your memories from Monday, if this is the phrase that's coming into the system, we run it through the STIE process. And when we do that, what happens is that we first of all, we standardize, then we split on whitespace to get individual words. Then we assign words to integers, and then we take each integer and essentially create a one-hot version of that integer.

And when we do that, basically, we have a vocabulary, right? And in this example, we just have 100 words. And you will note that this vocabulary which you arrive at once you standardize and tokenize has words like the because we decided not to remove stop words like a, and, the, and so on.

So just to be clear, standardization-- here, standardization-- while it has historically been all about stripping punctuation, lowercasing everything, removing stopwords, and stemming-- while that has been true historically, if you look at modern practice, people essentially strip punctuation, maybe, and then lowercase, and they often don't even bother to do stemming and things like that, or to remove stop words. And that's why in Keras, the default standardization is only lowercasing. And punctuation stripping. This detail may actually be handy for homework too, perhaps. That's why I'm pointing it out.

OK, so that's what we have. And so for each word that's coming in, we have a one-hot vector. The one-hot vector is just onto the vocabulary. And we can either add them up and get a multi-hot encoding, or we can-- sorry-- get a count encoding, or we can just do or. Look for just any ones in a column, and get multi-hot encoding. So that's what we saw last class.

But this scheme, while it's quite effective for simple problems, it has some very serious shortcomings. And so we will delve into those shortcomings and then step back and say, all right, is there a solution to fix these things? OK. Explain the problem with one-hot vectors. There are lots of problems. Any volunteers?

AUDIENCE: So similar words are understood differently. So for example, if you have this house is great and this house is awesome, great and awesome is encoded differently. And obviously, differently, even though this [INAUDIBLE].

RAMA RAMAKRISHNAN: Absolutely. So what he's pointing out is that if you have two words which are synonyms-- let's say great and awesome-- we would hope that the way we represent them using these vectors would have some connection to what the words actually mean. In particular, we would hope that if they mean similar things, that they are close by. If they mean very different things, we would hope that they are very far away. Things like that, common sensical expectations of what you want the vectors to have.

So it clearly won't have that, and we'll look into it in more detail in a bit. But before we do that, there is also a computational issue, which we covered last class. Which is that if the vocabulary is really long, then each token, each word that's coming in here will have a one-hot vector that's as long as the size of the vocabulary. If you have 500,000 words in your vocabulary, every little word that comes in has a vector which is 500,000 long, which feels like a gross waste of stuff.

Now, you can mitigate it somewhat by choosing only the most frequent words. But it does increase the number of weights the model has to learn, and increase the need for compute, and data, and so on and so forth. Now, let's say that we have created a vocabulary from a training corpus. We have a bunch of text that's coming in. We have done the SD. The standardization, tokenization, we've created vocabulary from it. And let's say we get the words movie and film.

So the question is-- and O's observation gets to this immediately. If you look at the words movie and film, are these two vectors close to each other or not? So if you have two vectors, how would we measure closeness? What's the simplest way to think about closeness? It's not a trick question. Distance. Yeah, exactly.

So if they're really close, distance-wise, you would hope the similar words should be close by. So here, let's just imagine that the vector for movie-- let's say your vocabulary is, I don't know, 100,000 long. So your vector is 100,000 long, and the word for movie is the position. So this has a 1, everything else is 0. Right?

Sorry, this is the vector for film, and maybe this is the position for film. So that has a 1. Everything else here-- 0. What is the distance between these two vectors? You just use the Euclidean distance. So the Euclidean distance, you will recall, you literally just take the difference of these values, square them, add them up, take the square root. So which means that all the zeros will obviously give you 0. This 1 is going to give you a 1. This comparison is going to give you another 1. 1 plus 1-- 2. Root 2. That's the answer. So this is between these two vectors, is root 2.

Now, so the distance between them is root 2. What about the one-hot encoded vectors for good and bad? Clearly, good and bad mean opposite things. What is the distance between the good and bad 0, 1 vectors? Still root 2. Because the zeros don't mean anything, the ones are not in the same place. So when you subtract the 1 and the 0, you'll get ones and ones. Add them up to root 2.

In fact, you take any two words in your vocabulary. What's the distance between the two one-hot vectors for those words? It's root 2. So if any two words have the same distance, does this even have a notion of distance? It doesn't. There's no notion of distance from one-hot vectors. It has no connection to the actual meanings of these words. It's just a way of representing them. So that is the big problem with one-hot vectors.

So the distance between them is the same, regardless of the words. It's got nothing to do with the meaning of the words. And this is a huge problem, which we'll have to solve. So to summarize where we are, if the vocabulary is very long, each token will have a one-hot vector that's long as vocabulary. That's a computational and training problem. And then this is a deeper problem where there's no connection between the meaning of a word and its vector.

So wouldn't it be nice if vectors that represent synonyms-- movie and film, apple-banana-- hopefully, they are close to each other. Wouldn't it be nice if the vectors for things that mean very different things were far from each other? So let's take a look at a particular example. OK?

Let's assume that we have been magically given these vectors so that they actually have some notion of meaning. And for convenience, let's say that we take just the first two dimensions of these vectors-- the first two dimensions-- so that we can do a scatter plot on them. So we plot the first dimension of these vectors, the second dimension. And what we have in this little cartoon is we have plotted the word for factory, for home, for building, and they all happen to be clustered here.

Clearly, this representation is capturing some notion of what the thing is, some sort of building And here, we have, a bicycle, truck, and car. Clearly, this is the automobile cluster, transportation cluster. And here, we have a fruit cluster. And here, we have sports balls cluster. OK? Because it's a cartoon, things are all nice and cleanly separated.

So now, if you take the word apple, where do you think it's going to go? It's going to go into A, C, D, or B? C, right? It makes eminent sense it's going to go to C. Good. Now wouldn't it be nice if, more generally, if the geometric relationship between word vectors represents a semantic relationship between the underlying objects that the words represent? And I say relationship and not distance because it's not just distance. It's actually more than that.

So let's look another one. Here, we have-- this is the vector plotted for puppy and dog, and this is calf. We have plotted the word for calf. And let's say that we need to figure out where would the embedding-- the word vector for cow appear? Where is the most logical? Should it be A? Should it be C? Should it be B? Where should it be?

AUDIENCE: This is C.

RAMA RAMAKRISHNAN: C? OK, what's the logic? Any volunteers? Just put your hand up. Yes?

AUDIENCE: A calf is a baby animal, whereas the cow is probably the adult. So it'd be closer to dog, which is B-- the adult version of the dog.

RAMA RAMAKRISHNAN: Got it. So you're basically saying, go from the puppy version to the grown up version. That's what you're getting at, right? And that's a totally valid way to think about it. But there are a couple of ways to think about this-- which, this is one of those two ways.

So what you can do is you can actually look at it and say, well, OK. If this is bringing you bad memories of GMAT and GRE and stuff like that, I apologize. So a puppy is to a dog like a calf is to a cow, right? Which means that there's exactly what Jay is pointing out. You can go from the baby version to the full grown version if you go in the horizontal direction.

But maybe, if you go in the vertical direction, you're essentially going up and down the young entities of animals. So here, you're growing with-- you're still across the same dimension of animals. You're just going from the age level. That is the band here. So this is the grown up version of a whole bunch of animals, the puppy version of a whole bunch of animals.

So the vertical dimension measures some variation across animal species of the same, roughly, maturity stage. So these directions also matter. It's not just the distance. That's what I mean when I say semantic relationship and geometric relationship. The relationship is distance and direction. Both have to be involved.

So now, word embeddings-- as we will learn soon-- are word vectors designed to achieve exactly these requirements. They will achieve these requirements, and they will fix both these problems very elegantly. So let's say that we have word embeddings that achieve both these problems. Are we basically done? Can we declare victory? Or is there anything that even words which actually capture the meaning of the underlying thing don't fully address? Is there any remaining problem we have to worry about? Yes?

AUDIENCE: Context.

RAMA RAMAKRISHNAN: Context? Yes. Context, right? The fact is a word's meaning-- sure, every word has a meaning. But we know that some words have multiple meanings, and that meaning is really inferrential or you can make sense of it-- only if you know the surrounding context.

If you see the word bank, B-A-N-K-- sure, it could be a financial institution. It could be the side of a river. It could be the act of a plane turning in one direction. It could be someone hoping for something, banking on something. The list of possible meanings of the word bank is basically enormous, and you cannot figure out what it means unless you know what else is going on around that word.

So context is super, super important. And these word embeddings just tell you what the meaning of the word is. And basically, what's going to happen when you have a word which could mean many different things, it's going to give you some average version of that meaning. The average version is not going to be very good. There are some words which only mean one thing, and you'll be OK there. But for the rest of it, it's going to be tough.

So what we need is some way-- you need to find a way to make word embeddings contextual. Meaning, we need to somehow consider the other words in the sentence. So if we can do that, then we will be in great shape to solve all sorts of NLP problems. So as it turns out, contextual word embeddings or word vectors or word embeddings will achieve both these requirements. They capture the semantic geometric relationship thing I talked about, and they are contextual. They're really fantastic.

And the key to calculating contextual word embeddings is the transformer. That is why transformers are justifiably famous. That's the lay of the land here. So today, we're going to look at how to calculate standalone or uncontextual word embeddings. And then starting Monday, we will take these un-standalone embeddings and make them contextual using transformers. That is the plan. Any questions so far?

So now, let's think about how we can learn these standalone embeddings from data. Now, the naive way to think about it would be, hey, why don't we manually collect a whole bunch of synonyms, antonyms, related words, et cetera and try to assign embedding vectors to them that satisfy our requirements? Now as you can imagine, there's going to be a long, painful, and never quite complete exercise.

And given that we are machine learning people, question is, can we do it in a better way? Can we just learn it from the data without doing any of this manual stuff? And the key insight that makes it all happen is this humble looking line on the screen by John Firth, who was a linguist. "You shall know a word by the company it keeps." I wish I could deliver this in a British accent.

"You shall know a word by the company it keeps." It's a very profound statement. And here is the key intuition behind this. It says, let's say that you have a sentence like, "the acting in the dash was superb." What are some words that you folks think are likely to appear in the sentence? Shout it out. Play, movie, show, musical. Those are all some great candidates. The acting in the movie, the film, musical, and so on and so forth.

Now, let's say that I ask you, what are some words that are unlikely to appear in the sentence? And I think we could all be here for days listing them out. I just listed these out. I love the word tensor, so I have to find a way to use it somewhere. All right. So the acting of the banana was superb. Clearly nonsensical.

So what we are seeing here is that if certain words are interchangeable in a sentence-- meaning, you change them, and the sentence still makes sense-- if they appear in the same context very often, i.e. if they're interchangeable, they are probably related. Sort of like, we don't even have to know what the word is. All we have to know is that this word and this word, you can drop them into a particular sentence. You can fill in the blank of that sentence with that word, and it actually makes sense. Then you're like-- oh, wow. OK. These words are related then.

You're inferring their relatedness, not by looking at them directly, but by seeing where they live. It's a very, very clever idea, and it'll slowly sink into you. So that's the first observation. If they appear in the same context very often, they are likely to be related. Generally, related words appear in related contexts. So all we have to do is to figure out a way to calculate context, and then use that to understand what the words are that happen to live in these contexts. And there are some beautiful ways to do these things, and we'll and we'll really dive deep into one such way to do it.

So what we're going to do in this approach is that since words that appear in related contexts mean same similar things, first of all, I have to define what do you mean by context? There are many ways to define context. We're going to go with a very simple explanation, a simple definition. Which is that if words happen to appear in the same sentence a lot, let me think that they are in the same context. So context here means sentence.

So what we can do is we can actually take a whole bunch of text-- maybe all of Wikipedia-- and then break it up into sentences. We'll have billions of sentences. And then for all these billion sentences, we can literally go and count for every pair of words how many times are both these words showing up in the same sentence.

And we call this co-occurrence. The words are co-occurring in the sentence. And it doesn't have to be next to each other. We know that in complicated words, a word at the very end of the sentence could actually alter the meaning Its meaning could be altered by a word that happened in the very beginning of the sentence, and it could be a really long sentence.

So we take the whole sentence and say, are two words co-occurring in the sentence? Yes or no? And we just count them up. And when we do that, we will get something like this. So this just captures what I've been talking about. Identify all the words that occur, let's say, in Wikipedia. And then for every sentence, you look at every word pair and count the number of times they appear in the same sentence across all those sentences.

This is a word-word co-occurrence matrix. So for example, let's assume that you took all of Wikipedia, looked at all the words, distinct words, and you found there were 500,000 words. So there are 500,000 words here in the columns. 500,000 words on the rows. So columns and rows. And then you go, and each cell of this table basically has a number that you calculate which is the number of times the word in the row and the word in the column happened to show up in the same sentence. That's it.

So for instance, if you look at deep and learning-- the word deep and the word learning-- maybe those two words occurred in the same sentence, maybe 3,025 times. 3,025 sentences across all of Wikipedia. You put 3,025 right in that cell. Many words are unlikely to appear in the same sentence. So much of this matrix is going to be 0. But we fundamentally form this co-occurrence matrix.

This matrix essentially embodies all the context information that we can work with in a very compact, beautiful, elegant way. And using this, we're going to try to figure out what the word embeddings actually are going to be. So by the way, the approach I'm describing here to calculate standalone embeddings is called GloVe. It's called GloVe.

And standalone embeddings first came on to the NLP deep learning scene-- there were two ways of doing it. One was called word2vec. Word2vec. The other one is GloVe. And they are both comparable. They use slightly different mechanisms of doing this. We went with word for this lecture because I think it's actually a little easier to understand, and equally effective.

So this is what we have. And so what we want to do is we want to learn these embedding vectors that can be used to essentially approximate this matrix. If we can find vectors that can actually approximate this matrix, then hopefully those vectors do, in fact, capture some notion of what the words actually mean.

So let me put it differently. You come to me with this matrix. And you say, OK. Rama, do you have embeddings for me. And I'm like, yeah. I reach into my bag, and I'm like, OK. Every one of those 500,000 words, I have an embedding. Let's ignore for a moment how I actually calculated the embeddings. I have the embeddings. How will you know if my embeddings are any good? How will you know? How can you actually assess if the embeddings are any good?

Well, you can certainly say, OK, give me the embeddings for movie and film, and you can see if they're really close by. If you look at the embedding for movie and tensor-- and hopefully they'll far away. But you'll never get done, right? How can you systematically evaluate this?

Well, what if you could actually-- what if I come to you and say, not only am I going to give you an embedding, here is a procedure which you can use with these embeddings to validate how good they are. And here is the procedure. What you can do is you can use the embedding to recreate the co-occurrence matrix. And if the recreated co-occurrence matrix actually matches the real matrix, well, these embeddings probably are pretty good.

Remember, the whole point of the co-occurrence is to handle this context information. So if the embeddings can actually recreate them, reconstruct them pretty close, right? It will never be perfect. But if it comes pretty close, they'll be like, wow, OK. These embeddings do mean something.

So if it turns out, for instance, that the matrix has 3,000 possible value of 3,000 for deep end learning and values of, say, 50 for extreme learning-- under embedding comes in and says 3,002 for the first one and 48 for the second one, we'll be like, we'll be pretty impressed. [INAUDIBLE] needed to be that close unless it was actually capturing something. So that's what we're going to do.

And so we're going to take this logic of saying find embeddings that can approximate what we actually see in Wikipedia. And we're going to use that idea to actually build a model and learn from it using nothing more than basically linear regression. And here, you are thinking the linear regression is useless now that you've graduated to machine learning, right?

So we can think of the embedding vectors that we want to figure out as just the weights in a model in a little regression. We can think of the co-occurrence matrix as just the data we're going to use in this model to estimate these weights. And the model we're going to use is something like this.

So first, I have to inflict some notation on you. We will denote the co-occurrence matrix of, say, words I and J as Xij. Xij is just data. It's just data, OK? It's not a variable, it's data. And then we'll denote an embedding vector for each word. So we're going to calculate a vector for each word. So we'll call it wi. wi is the embedding vector for each word. And we will also assume that some words are just inherently very popular. They're going to show up all the time, like the word the.

So we'll assume that every word has some natural frequency of occurring, like movie versus flick, the versus tensor. So we want the vectors to capture the co-occurrence patterns independent of how naturally frequent the words are. And so to capture this natural frequency, we will assign a bias, or bi, to each word that we're going to calculate. And all this will become clear in just a moment.

So with this setup, basically what we're saying is something very simple. We're saying, look, this co-occurrence matrix that we have that we're able to compute-- it came about because in truth, in reality, in nature, there are these embedding vectors for every word. There are these biases, bi, by for every word. And every co-occurrence number that you see just came about because under the hood, mother nature grabbed the bias number for the word i, the bias number for the word j, took the two embedding vectors which only mother nature knows at this point, did the dot product of them, added them, and that's how we get this number.

So it basically says the number you see is the sum of the inherent popularity of the first word, plus the inherent popularity of the second word, plus the way in which these two words connect to each other. That's it. And you will agree with me that it literally can't get simpler than this. If I tell you, hey, here are two things. I want you to tell me how connected they are. You'll be like, well, let's take the first one. Figure out how inherently popular it is, inherent popularity. And then, of course, you to worry about the connection, so do a dot product. That's it. Those three things.

So this is what we have. Now you may have seen from your good old linear regression that whenever your dependent variable happens to be positive-- guaranteed to be positive-- and it ends up having a big range, we always advise your folks to take the logarithmic transformation to squash it into a narrow range because that will make these models much more well-behaved.

Regression if the y value is like a huge range. The canonical example is that if you are trying to model the net worth of people, it's going to have a long right tail with people like Elon and Jeff and so on the right side, and the rest of us on the left. So to model this big, long tail distribution, you just take the logarithm, just squash everything to a very narrow range. And that will make regression much better behaved.

Here, most of the counts are going to be 0, but some of the counts could be very high. And therefore, we wanted to-- if you take the logarithm, it makes it much better behaved. So take the logarithm here. So this is actually our model. That's it. And I know that many of the numbers are 0, and log of 0 is not defined, so we can just add the number 1 to all the numbers to avoid that kind of technical arithmetic problems. But this, conceptually, is what's going on. This is the model we want to calculate.

So given that we have essentially postulated this model and we have this data, this co-occurrence matrix, how can we actually find the weights? How can we actually find the b's and the w's? What will what should we do? Go back to the fundamentals of regression. Think about it conceptually. You have some model, which has some weights. There's some data you can use to train the model. And you need to find the best set of weights. What does the best mean here?

AUDIENCE: Probably the lowest error.

RAMA RAMAKRISHNAN: The lowest error, exactly. There are many ways to measure error. What is the simplest thing we could use? So what you would do is you would actually do mean squared error, which is what you're getting at. You could take the actual thing, you could take the predicted thing, take the difference and square it, and minimize the sum of it.

If your model exactly nails every number in the co-occurrence matrix, the rest is going to be 0. So what we do is we literally just do that. This is the data. This is the actual predicted value. Predicted value, actual value, difference squared, add them all up, minimize. OK? Yes?

AUDIENCE: I'm kind of lost where how is this capturing the context? Because unless my input data is having that context, how will this actually differentiate based on where the particular word is used?

RAMA RAMAKRISHNAN: The way the word is-- so let's take two words, like deep and learning.

AUDIENCE: No, let's take one word and change it according to the final.

RAMA RAMAKRISHNAN: OK. Sorry, go ahead.

AUDIENCE: Yeah, so basically let's say I have a banana. So it's a fruit in some context. And I could be saying, he's going bananas. So whatever, right? So now these are two different contexts in my understanding. And my same model needs to be able to tell me that banana is the right word in this context, but wrong word in this context, or correct in both contexts.

RAMA RAMAKRISHNAN: Yeah. Very good question. So let's actually spend a minute on that. Good question. I'm going to swap to my iPad. So let's assume that this is our co-occurrence matrix. And then we have words going from all the way to-- let's say zebra. These are all the words in the vocabulary, and we have a through zebra here. And now what we have is we have apple and banana.

So basically what's going on at this point is that here, every number here measures for every word here, how many times that word and apple show up in the same sentence. It is not measuring, to your point, how many times apple and banana are showing up. It's measuring how many times apple is showing up in each sentence.

Now, if apple and banana are interchangeable, what do we expect these two rows of numbers to look like? Let's assume that apple and banana are perfect synonyms, just for argument. Let's say it's a perfect synonyms. What do we expect these two numbers to look like? Very similar. So if two words are related, their entry row vectors, the co-occurrence matrix are going to be very, very similar. So that is how the context comes into the co-occurrence matrix.

So what we want to do is we want to find if embeddings can recreate the same pattern of numbers in these two rows, it's actually capturing the underlying context. So words which are similar will zig and zag together the same way through the co-occurrence matrix, and that's where it comes in. yeah

AUDIENCE: What's on the diagonal of the co-occurrence matrix? Like, if apple shows up twice?

RAMA RAMAKRISHNAN: Oh, I see. So here, you can just ignore the diagonal, typically, because all the action is off diagonal entries. So that's basically the idea. And if words which are very similar will have a very similar pattern of numbers, and then any embeddings that can actually recreate the same pattern of numbers is capturing the underlying reality of what's going on.

If words are unrelated, those two vectors-- let's say that the word you have is-- let's assume the word is-- you know what I'm going to say. Tensor. These two vectors won't have any connection to each other. Which means if you look at something like the correlation of those two vectors, it's going to be around zero.

But words which are interchangeable will have a very high correlation. Words which are antonyms and never show up in the same place together may have a highly negative correlation, close to minus 1, for instance. So that's the intuition behind what's going on on these row vectors.

And so the point is given this co-occurrence matrix is capturing all these word-word correlational structure, any embedding that can recreate it must have captured the structure as well because you can't recreate something like this with great fidelity unless you have some notion of what's going on under the hood. That's the basic idea. Yeah?

AUDIENCE: So just connecting just from this question-- so in that example then, banana is a fruit and apple is a fruit as well. Banana and apple are synonyms. And you are going mad or you are going bananas, and that comes together? Is that how we read it?

RAMA RAMAKRISHNAN: Oh, I see. You're going mad. You're going bananas. Yeah, so those will also have some correlation structure to it, which the embeddings will hopefully catch. But words like banana, which are very-- the thing is it's called polysemy, where the word looks one way. It looks the same. It's like the word bank, right? It can mean very different things in many different contexts. So the embedding is going to be some average representation of it. But we're not happy with that average, and we will get around that average next week when we do contextual stuff.

All right. So that's what we have here. So to go back to this thing-- so what we can do is-- yeah?

AUDIENCE: After getting that initial, how did we get the mean square error in this? Because we split the embedding you put in the data and then we get...

RAMA RAMAKRISHNAN: We haven't calculated the embeddings. We are trying to calculate them. Those are just-- it's like in regression, you have beta 1 times x1 plus beta 2 times x2 kind of thing. The betas are what the regression produces for us. The embeddings are exactly that. They're just coefficients that we're trying to figure out. The data is only the x's, the Xij.

And so this is what we're trying to calculate. And so what you can do is you can actually start with some random values for these things and then keep on trying to improve to minimize the error starting from these random values. Are you aware of any algorithm that allows us to take random values starting point and then minimize some notion of error?

AUDIENCE: Who takes the random? How can we order it random if it has to be random?

RAMA RAMAKRISHNAN: How do you know it's actually random? Oh. So that's actually a very deep question. And so it's actually a tough question because ultimately, the random number is coming from a computer. And we know how the computer runs. It's deterministic, at the end of the day.

So we actually use something called pseudorandom numbers. And there's a whole specialized field of math which essentially says, look, how can I get random numbers that are sufficiently random, even though they come from a non-random computer deterministic process? So we can talk offline about it. But fundamentally, all these systems have some random number generators built in. We just cross our fingers and hope for the best and just use them.

So come back to this. We can start with random values for these weights, and then we can try to minimize the squared error. Are you folks aware of any algorithm that can help us do that? Yes?

AUDIENCE: Gradient descent.

RAMA RAMAKRISHNAN: Yes, gradient descent. Again, comes to the rescue. And since we are cool, we'll do stochastic gradient descent. So that's it. So gradient descent actually doesn't care what the function is, as long as you can calculate a derivative from it. As long as you calculate the gradient, you're good. So we can just run gradient descent on this thing.

One key point here is that stochastic gradient descent work for any models. As long as you can calculate good gradients from them, it doesn't have to be a neural network. Any mathematical function, as long as it's differentiable and gives you a good gradient. So here, this is not a neural network, per se, but we can use gradient descent for it.

So we do that, and when we are done, we would have calculated some nice embeddings. We can also calculate all these biases. But we don't need the biases anymore. We can just throw out the biases because we only care about the embeddings and how they connect to each other. Yeah?

AUDIENCE: So when you're doing the regression, are you predicting the co-occurrence matrix?

RAMA RAMAKRISHNAN: Mm-hmm.

AUDIENCE: OK.

RAMA RAMAKRISHNAN: Exactly. So actually, let me just show a very quick numerical example here. So let's say, for example, that-- you know what? Deep learning. So this is, say, w1 and this is w2. And this is the vector. And let's assume for a moment that it has two dimensions. Two dimensions. And we also need to calculate b1 and b2, which is just a number. And let's say the number for deep learning in the co-occurrence matrix, let's say it has occurred 104 times.

So all we are doing is to say log of 104-- that is the actual value-- minus b1, which we don't know, plus B2, which we don't know. And then this thing here, let's just call it w1 one, w1 two, w2 one, w2 two. And then we're just doing the dot product, which is w1 one times w1 two plus w2 one, w2 two.

So this is our prediction. Where is that cool laser pointer? Yeah, so this is our prediction. This is the actual. So all we do is to say, OK, this thing-- the difference, we're going to square it. And then we're going to do the same exact thing for every other word pair. And when we are done with all of that thing, we just take this whole thing and say gradient descent, minimize. So then it has to find the b's and the w's and everything for every pair, every word. So that's actually what's going on. Make sense?

All right. So by the way, here, I said let's assume that the embeddings are just vectors which are dimension two. Well, that's an arbitrary decision that I made just to show you how it works because I was doing it by hand. But more generally, we get to choose how long these vectors are.

And the longer the vector, the more interesting ways it can actually reproduce the co-occurrence matrix. It has more flexibility. But the longer the vector, what is the risk that you run? Overfitting. Because these are all parameters, at the end of the day. More parameters you have, the more risk of overfitting. So you get to choose how big these things can be. Yes?

AUDIENCE: Don't you find it surprising that we're able to fit a model where we have a lot more parameters than we have data? Because usually when it comes to machine learning, we like sparsity. We like to not have a lot of parameters. But here, we're going to have, as you said, the number of dimensions times more parameters than we have of data points.

RAMA RAMAKRISHNAN: Well, in this particular case, as it turns out-- let's assume that you only have 10 words. And for each word, let's assume that you have-- just to keep the math simple, you have a two dimensional vector. So 10 words times 2, that's 20. Plus you have 10 biases for the words, so there's another 10. That's 30. But 10 by 10, the matrix has 100 entries. So because of the matrix being order n squared matrix, you'll have a lot more numbers than parameters. In this particular case, you have more data than parameters.

So that particular problem doesn't apply in this case, but that does show up in other cases. And there is some very interesting research in neural networks which suggests that oftentimes, the traditional assumptions of data and overfitting and all can all be called into question under some situations. I'm happy to tell you more offline, but if you're curious, just Google something called double descent to know what I mean. But in this case, it's not a problem.

OK. So what that means is that we can choose how big these things are. So if you look at one-hot vectors, where there's a 1 and everything else is 0, depending on the position of the word-- these are vectors as long as the vocabulary, as we saw earlier. Word embeddings, on the other hand, they can be very dense. The numbers that make up these embeddings, we actually want to figure out from the data what they are. So it can be anything.

So the first dimension may stand for some combination of brightness plus speed plus animalness or something. We have no idea what it means. All we know is that it's able to reproduce the co-occurrence matrix really well, so it probably has figured something out. And so we can keep it really short. So the word embeddings tend to be very dense, meaning not zeros and ones, but some arbitrary numbers. It's very low dimensional, and it's of course learned from data.

So once you do this, once you actually run GloVe on this data and do gradient descent and so on and so forth, you actually come up with embeddings. And then you can actually plot the embeddings. You can take these embeddings and just plot them. Here, they are not literally plotting the first two dimensions. They're using a particular technique called t-SNE, which is a way to take long vectors and project them to 2D space for visualization purposes.

And you can see here, some very interesting things are showing up. So they're basically, they plotted the embedding for brother-- nephew, uncle, sister, niece, aunt and so on and so forth. It's all showing up here. This is the embedding for man, embedding for woman, sir madam, heiress, heir, duke, emperor, king. You get the idea.

So clearly, there are patterns here where things which are similar in their nature are all hanging out together in the same part of the space, which is comforting, which is good to know. But as I mentioned earlier, it's not just about the fact that similar things happen to be near each other. The direction also actually matters. And beautiful things happen when you look at directions.

So for instance, let's say that you have man, and you want to go from man to brother. So to go from man to brother, you have to start with man and then travel along this arrow to get to brother. So this arrow has some notion of a person becoming a sibling. So you would hope that if you take that same arrow and then start here with that arrow, hopefully the woman will become a sister. Sure enough, it is.

So this is called word vector algebra. Embedding algebra. And these relationships are actually showing up in the data. We didn't tell it any of these things. We just literally gave it the co-occurrence matrix and asked her to reproduce it. So I find it pretty shocking that these things are actually true. And it gives us evidence and comfort that whatever has been learned does have some deep connection to describing the underlying nature of what's going on. It's not some statistically fluky artifact. Yeah?

AUDIENCE: So we can find synonyms by context or by adjacency to other words, and not by replacing the same word, right? Because they won't appear in the same sentence if you have two words--

RAMA RAMAKRISHNAN: Right. They won't appear in the same sentence, but the pattern of co-occurrence will be the same for them, which is what we've been able to reproduce with these embeddings. So that's the key idea.

AUDIENCE: So my question is around how are we able to capture all these directions and in 2D matrix versus a multi-dimensional matrix? Because I feel like, OK, so this relationship is kind of concerned. Yeah, I'm going to a family or blood relationship or something of the sort. But how does it not mess up the other side of that matrix?

RAMA RAMAKRISHNAN: No, this is just a visualization thing. So we're basically taking this-- as you will see, GloVe embeddings come in lots of different sizes. And this, I think, uses the 100 dimension embedding and just projects it to 2D space using a particular technique, and then looks to see what's going on. Yeah?

AUDIENCE: If the input data being the co-occurrence matrix is biased, are we amplifying that bias?

RAMA RAMAKRISHNAN: Yes, we are. Yes. No, it's a great observation. Any data you scrape from the internet and use for this modeling exercise will be subject to all the biases that produce the data in the first place, and the model will faithfully learn those biases. And if you're not careful, it will perpetuate them. And that's a whole very important topic that unfortunately we won't cover in this course because of time constraints. But it's something you always have to worry about when you're building these models.

AUDIENCE: How do we think about the dimensionality of the embeddings, not the 2D representation of the actual embedding?

RAMA RAMAKRISHNAN: The one that we choose that's in our hands? So you should think of them as a hyperparameter. So much like the number of hidden units to use in a particular hidden layer, it's a hyperparameter. I would, again, start small. And if it solves the problem that you're trying to solve with these embeddings, great. If not, keep increasing them. And at some point, there might be a flattening out and an overfitting dynamic, and then you stop. So just think of them as a hyperparameter. Yeah?

AUDIENCE: Do you see any benefit in practice of using penalized regression to do this instead of making the embeddings more sparse or just lowering the magnitude of them?

RAMA RAMAKRISHNAN: Yeah.

AUDIENCE: [INAUDIBLE]?

RAMA RAMAKRISHNAN: Yes. So there are lots of techniques to apply regularization in the estimation itself of all these numbers. I'm happy to give you pointers. I'm just going with the simplest version possible. Yeah?

AUDIENCE: I'm not understanding why overfitting is a problem in this case because we're not doing any out-of-sample predictions. Wouldn't you want the embeddings to be high dimension so you capture unique relationships?

RAMA RAMAKRISHNAN: Ah, interesting question. So the question is, given that there's no notion of a out-of-sample test set that we're going to evaluate these things on, why do we really care about overfitting? Shouldn't we do the best we can to capture everything in the data?

Well, the thing is even when you're not trying to use it for out-of-sample prediction, you do want to make sure that your model only captures the true patterns and not the noise. In every data set, there is always noise, and you want it to capture the signal but not the noise regardless of what you use it for. Because if it captures the noise, then the insights you draw from the word embeddings may be flawed. That's the reason.

OK. All right, so let's keep going. So here, the algebra is brother minus man plus woman is sister. That's it. Human biology reduced to a single sentence. All right. So now the pros and cons of these things are you should use something like a GloVe embedding if you don't have enough data to learn a task specific embedding for your own vocabulary.

As I'll show you in the Colab, you can actually learn these things just for your own data set, if you want. You don't have to use these GloVe embeddings. But the reason to use these pre-trained embeddings is that if you're working with natural language, the word is the word. It means something. And so there's no reason for your model, for your little use case for you to actually somehow learn all the fundamentals of English. The fundamentals of English are the fundamentals of English. May as well learn it once and then piggyback on it.

So that's the whole idea of using pre-trained embeddings because these things are all common aspects of language. May as well learn them using all the data you can throw at it, and then you can fine tune and tweak and adapt to your particular use case. And this is particularly useful when you don't have a lot of data in your particular use case. That's one big advantage.

Now, it does have the drawback that this embedding will not be customized to your data. For example, if you're trying to build an application for a medical or legal use, it's going to have a lot of jargon. And this pre-trained embedding trained on all of Wikipedia may not capture enough of the jargon and know its meaning really accurately.

So what you want to do is you want to take this thing-- you may still want to take this thing, and then you can adapt and fine tune it using your jargon packed, heavy, domain-specific data set. Those are some of the things to keep in mind. And of course, we can also learn it from scratch if you want. And the Colab, I demonstrate all these options.

So when you're working with embeddings in Keras, so what we do is remember STIE, where after we standardize and tokenize and index, at this point, we go from integers to vectors. And so far, we have been using integers to one-hot vectors. Here, we're going to use embedding vectors that we're going to learn or that we're going to use from GloVe.

And so what we do is we tell Keras's text vectorization layer to do only S, T, and I. And then we will use a new layer called the embedding layer to do the encoding. That's how we're going to divide it up. So we'll take a look at this first before we switch to the Colab.

So before, we told Keras in this layer, output mode should be multi-hot or whatever. Here, we don't want it to actually encode anything in multi-hot. We just want to give it integers back. So we tell it, give me int. That's the first change. We tell it, give us int. If you say give us int, it'll stop with STI and just give you the integers.

And then what you do is that all the incoming centers are going to have different lengths. So what we want to do is we want to actually take all these sentences and normalize them so they are of the same length And the way we do that very quickly is that we choose a maximum length for the sentences, and then if something is exactly fits that length-- perfect.

Let's say in this case, we want a max length of 5. Cat sat on the mat is exactly five. Boom. Fits perfectly. But if something is smaller-- I love you is only three of these things-- we actually pad it with something called the pad token. Also the unk token-- pad token is a special token which we use for padding. And so in Keras, you will see, will use zeros for these paddings so that it fills it up and gets all the way to the end. And if you have something which is much longer than five, you just truncate everything else and just use the first five. So this is what we do to get all the sentences to be of the same length.

And once we do that, we then go to the embedding layer. And the embedding layer is actually very simple. What is an embedding? It's just a vector. And we need a vector for every token. Of course, we're going to learn these vectors. We need one for every token.

So in this case, for example, let's say that these are all the tokens we have in our vocabulary after the STI process. Maybe in this case, we have 5,000 tokens. Each token, we have this embedding vector, and we choose what the dimension of the embedding vector is right. And so we can set it up by saying keras.layers.Embedding.

And we tell it max tokens which means how many rows do we have here? What is the vocabulary size that we are working with? And then we tell it, OK, this is how long I want each embedding vector to be. So rows, the size of the columns, and that's the embedding layer. And we will use it in a second. I just want to show it to you here because it's slightly clearer.

So when an input sentence arrives, the text vectorization layer will run STI on it. It will truncate and pad it to max length as needed. So let's say this phrase comes in. STI will give you the same tokens plus pad pad because let's say the max length is 5, and then these are the corresponding integers. And then the embedding layer will just look up the corresponding vectors.

So for example here, the vectors are-- we need to look up the vectors for 23, 9, 5, 0, and 0. So we just go here and look up 23, 5, 9, and 0. And then once we have that, boom, this is the resulting output. So whatever input sentence comes in, we have now five embedding vectors that have been looked up from the embedding layer.

And once we do that-- this is a table. So I love you comes in. It becomes this table. As we have seen before, neural networks can only accommodate vectors as inputs. We need to make this into a vector. And as we have done before, we can either take all these things and concatenate them, make a one long vector, or we can find a way to average them or sum them and things like that, as we have seen before. And we will use the same-- the simplest thing is probably just average them.

So these are some options, but we will average them here. And this is called the GlobalAveragePooling layer, 1D. And all it does is whatever you give it-- a table you give it, it just takes each dimension and averages it. The first dimension average, second dimension average, and so on and so forth. And once that's done, that's the whole flow.

So the phrase comes in, STI gives you these things, padding as needed or truncating as needed. We look up the embeddings from the embedding layer, and then we get all this thing. We do global pooling on it, and it's done. The resulting thing is a vector that can then be passed into hidden layers just like we normally do. I'm going go with this a little fast, but make sure you look at it afterwards and understand every step, and the Colab will mirror this perfectly.

All right, so let's switch to the Colab. OK. All right. Can folks see this OK? All right. So we'll do the usual. Import all the stuff you need. And then because I want to plot some of these loss and accuracy curves just to see what's going on, I'll just bring in the functions from the previous Colabs here. And I think I already have downloaded this. Let me just make sure I have it.

It's not there. So let's do it again. This is the same songs data set that we looked at on Monday. So roughly 49,000 examples, as we saw before. We'll one-hot encode them. All right. So there's a bunch of stuff that we've already covered in class. So this is the thing. This URL has all the GloVe vectors available for download. I downloaded it before class because it takes a few minutes. And I've also-- did I unzip it? Yes, I did. So let's just look at the first few.

All right. So these are all the first few. We'll create an easier to view version of these GloVe vectors. So I'm going to use the vectors which are hundred long, but it comes in many different shapes. So we have 400,000 vectors. 400,000 word vectors each as 100 dimension. And these all have been calculated from Wikipedia using the model we described using gradient descent. OK?

All right. So this is the vector for the word for movie. I don't know what these dimensions mean, but there's something going on. It has figured stuff out. But the proof is in the pudding. So all right, now we will first set up the text vectorization and embedding layers like we saw before. And so I'm going to use a max length of 300 for the songs because all the sentences have to be the same length.

And you might be wondering, OK, why did you pick 300 or not, say, 400 or 200? So typically, what you do is you actually look at the length distribution of the songs you have, and you will find-- you're looking for an 80-20 or one of those things. And in this case, it turns out 90% of the songs have less than or equal to 300 words in our data set. So I'm just going to go with 300. It's pretty good.

The problem is if you actually say-- if you look at the song which has the maximum length, that might be like 3,000 words, and there might be hardly any songs of 3,000 long. You're just wasting a lot of capacity by doing that. So you're just being a little pragmatic here.

OK, and then as before, for the vocabulary itself, we tell Keras use the most frequent 5,000 words when you're doing the STI. So we do that, and we tell it the output mode is int, like we saw before. We have there. OK, perfect. OK, this is a very dangerous thing, where somebody is remotely changing it in another tab somewhere. Fingers crossed, OK? Take this one.

OK. So we have this, and this is what we did with all this stuff, as I've covered. So now we will adapt this layer, as we have seen before, using all the lyrics we have. Then once we do that, we'll take a look at the first few.

And so here's a very important thing. Before-- when we asked it to do multi-hot encoding and so on, on Monday-- the zero, the first position was unk. Unk had zero. But here, unk actually has one. And the reason is that the zeroth position is going to be used for essentially-- you can think of this as the empty string. That's how Keras will print out pad. So the zeroth position is the padding, the pad token. The first position is the unk token. So it's an important thing here.

So let's say that we do HODL, you're the best. We vectorize it. Do you think HODL is going to be part of those 400,000 word vectors [INAUDIBLE]? Not yet. All right, so let's try that. And as you can tell, HODL is an unknown word. That's why it's showing up here. So 1 is unknown. The index value 1 is unknown. 0 is pad.

But then this is unknown HODL, I-- sorry, you are the best. And then everything else from that point on is a 0 because we are padding all the way to 300. So that's why you see all these zeros here. All right, now let's just run everything through the vectorization layer, and then we'll get to the embedding layer.

OK. Now we will first-- just a bit of Python housekeeping to create a nice, easy to look at matrix. So what we're going to do is we're actually going to create a nice matrix that shows us all the GloVe embeddings. And so this is the embedding matrix, and this matrix has only 5,000 words, and each is 100 long. Why is this embedding matrix only 5,000, even though we downloaded 400,000 vectors?

So clearly, the 5,000 we used, there has some bearing to this. But what is the 5,000? We told Keras to take the most frequent 5,000 words in our corpus so we'll only have 5,000 in vocabulary. That's why there's 5,000. So we grab just the GloVe vectors for those 5,000 that Keras has chosen to be in the vocabulary, and that's our embedding matrix.

And then if you look at the first few rows, the first two rows should be all zeros because it's pad and unk, which clearly, GloVe doesn't know about. So it's all going to be all zeros. And so you can see all these zeros here. And then from the third onwards, you start getting some numbers. OK? All right. Next, we'll set up the embedding layer.

So basically, what's going on here is we tell the embedding layer how many rows, which is just the vocab size max tokens. What is the embedding dimension? Well, that's going to be 100 because the GloVe vectors are hundred. And then here's the thing. You can tell it in this embedding layer, just use this matrix I'm giving you as the embedding layer because we already know what embeddings are. We downloaded it from whatever GloVe.

So we will tell it to use GloVe as the weights for here, as the embeddings here. So we initialize it using that embedding matrix. And then we tell it, when we do back propagation later on, don't change any of these weights because somebody spent a lot of money to create this weights for us-- Stanford. So we don't want to further change them, just freeze them and use them as they are. And this mask zero business, I'll come back later. Don't worry about it for the moment.

All right. So once we do that, we are ready to set up our model. So this model is pretty simple. Keras input. The length, of course, is the length of the sentence, which is 300 long. And then the input runs through an embedding layer right there, and out comes a 300 by 100 table. And then we average pool it, and that becomes 100 element vector. And then we are back in familiar ground.

And we run it through a dense layer with eight ReLU neurons. Eight ReLU neurons. And then we run it through the final output layer, which is a three way softmax as before-- hip hop, rock, pop. And then we tell Keras that's the model, and then we summarize it.

OK, so that's what we have. And you can see here, the total parameters are 500,835, but the trainable parameters are only 835. It's because the total parameters are all the GloVe embeddings plus the things we added to the GloVe embeddings like the hidden layer and so on. But the GloVe embeddings are-- we have told Keras, freeze it. Do not train it. Which means only the rest of it is going to be trainable. That's the 835. Yeah?

AUDIENCE: So when we do the global average pooling, don't we lose any sense of meaning that we gain from the embedding, as we average very different embeddings together?

RAMA RAMAKRISHNAN: Sorry, say that again. I missed the first part.

AUDIENCE: If we average the embedding of apple and learning, for instance, you have very different words that are used in different meanings. So we have different embeddings, but we average it so we can't lose this--

RAMA RAMAKRISHNAN: We will lose a bunch of stuff. Yeah, yeah, yeah. So yeah, anytime you average anything, you're going to lose some nuance and so on. So the real question is despite that averaging, is it good enough for you? And sometimes it's good enough. Very often, it's good enough, as it turns out.

But as you will see when we go to contextual embeddings, there's just a better way to do it, when you have contextual embeddings. But it requires bigger models, more powerful stuff, and so on and so forth. And that's where you're going from the foundations to the advanced stuff. Yeah?

AUDIENCE: When we're doing optimization-- like, let's say we are having a word problem-- it's often best to optimize everything together than to optimize one part of the system and then optimize the response system. So in that case, why wouldn't we want to also change the embeddings? I understand why you would like to start with those weights that some people have spent a lot of money trying to find, but will we be able to find more specific embeddings related to our problem if we let everything be trainable?

RAMA RAMAKRISHNAN: Absolutely, absolutely. And in fact, you will see in the Colab that we will do that next. I just want to show people you don't have to do it. You start with not training it because it's going to be much faster, and then you train everything and see if it gets better. And sometimes it will get better, in which case it's great. Sometimes it won't get better.

And I will also show you-- and I probably will run out of time, which I'll do it on Monday. I will also show you, hey, what if you want to do your own embeddings from scratch without using GloVe? So all possibilities will be covered.

Yeah. So to come back to this-- this is the model we have. And then-- all right. So if we take a look at the first few embedding vectors-- by the way, this model.layers will give you every layer as a list of all the layers, and then you can just grab any layer you want and look at its weights. It's very handy.

So we are looking at the weights, and you can see here the first two vectors are all zeros because that stands for unk and pad, and then we have everything else. So everything looks fine so far. And now we just compile and fit it. So as usual, Adam cross-entropy accuracy, and then we will just fit the model. All right. It's going to take a few minutes.

And while it's running, so what you will see in this Colab is that in this particular case, the embeddings actually don't help a whole lot. Why do you think that is? Part of it could be because we're averaging a lot of stuff. Maybe that's hurting us. Yeah?

AUDIENCE: I mean, I think that embeddings are probably trained on some corpus, right-- like Wikipedia, or something like that-- that is a little bit different from the language we tend to use in song lyrics. And so maybe it's not its ability to extract the meaning of candy from a song lyric, maybe it's limited because it's thinking of all the other ways that could be represented.

RAMA RAMAKRISHNAN: Yeah, so there could be a mismatch between the corpus on which the pre-trained stuff was trained on versus the corpus that you're working with right now. That's one big reason. The other reason is that we actually may have-- we have 50,000 examples, basically. It's a lot of data. So when you have a lot of data, you may not need any of these things. These things tend to do really well when you don't have a lot of data, which means you get to piggyback on what these embeddings have learned from all of Wikipedia.

So when you have a smallish data set, basically the rule of thumb here is that when your data is really small, try to use a pre-trained model. And that's what you saw with the handbags and shoes classifier. We had 100 examples of handbags and shoes, and we used ResNet to basically get to 100% accuracy. The same of logic applies here.

All right. So here, let's see what's happening. OK, it's done. So we'll plot. Right. OK. Let's look. A very well-behaved loss function curve. OK. So there doesn't seem to be any massive overfitting going on. They're moving really nicely in lockstep. Let's see what the thing is.

OK, 63%, which is not great. It's not as good as what we saw before when we used all 50,000 examples and just trained something from scratch. And that's just because in this case, we have lots of examples. These pre-trained embeddings aren't as helpful as they could be. But if you have a small data set, they could be very helpful.

And now we go to what he pointed out. Why can't we just optimize these embeddings too? Why do we have to treat them as sacred? Let's just unleash back prop on it and see what happens. So we'll do that.

So here, what we do is we retrain it. But here, we set trainable equals true for the embedding layer. OK? This is the key step. Trainable equals true. Otherwise, it's unchanged. And then skip that. We'll run it and see what happens. So before, it was whatever, 63% accuracy or something. We'll see if it gets better if you train the whole thing.

And the thing is you can never be sure because it may start to overfit, which is why you just have to empirically see what's going on. There are no guarantees. All right. Any questions while it's training? Yeah?

AUDIENCE: In the first graph, when you had the training accuracy still increasing, that might suggest that you could use even more [INAUDIBLE].

RAMA RAMAKRISHNAN: Correct. Exactly, exactly. So in that curve, we saw that the training was continuing to increase. Typically, what's going to happen is that training will continue to get better the more you train it. The key thing is the validation also improving. If the validation continues to improve, there is a little bit more gas left in the tank. You can keep increasing more. If it starts to flatten-- and even worse, if it starts to go down, then you want to pull back. Yeah?

AUDIENCE: So you had used the max tokens to limit the vocabulary to the most common 5,000, and then the width of that was 100. What is the 100?

RAMA RAMAKRISHNAN: The 100 is just the length of the GloVe vector.

AUDIENCE: Does that mean that it can only capture how that word is related to 100 other words?

RAMA RAMAKRISHNAN: No, no. Basically, we are saying that every word, its intrinsic meaning can be captured using a vector of 100 dimensions. Each of those dimensions mean something. We don't know what it is. The first dimension could mean color. The second could mean some sort of location. The third could mean some sort of time of the year. We just have no idea.

AUDIENCE: OK. And the pre-trained model-- because we're not allowing it to learn the pre-trained model, it has those already. We don't know what they are, but it has some--

RAMA RAMAKRISHNAN: The people who created it don't know what they are either.

AUDIENCE: Yeah.

RAMA RAMAKRISHNAN: All they know is that for each word, they learned 100 long vector. And that 100 long vector was able to recreate the co-occurrence matrix. And then they probed it using that visualization of man, woman, sister, brother, all that stuff, and it seems to fit with what we would expect.

AUDIENCE: Can you think of it as analogous to-- when we did the convolutional ones, you have the number of kernels. So in this case, so if you had 32 kernels, it's like 32 things it can learn.

RAMA RAMAKRISHNAN: I think that's actually a great analogy. I love it. That's a great way to think about it. Yes. Much like we got to decide how many filters to have, here, we get to decide how long the embedding dimension needs to be. And our hope is that the more things we are able to accommodate, the more complicated things it will pick up. At the same time, you don't want to have too many of these things because it's going to start picking up noise, and that's never a good thing. OK. Was that a question on this side? Yeah? Go ahead.

AUDIENCE: [INAUDIBLE]. Why do we use embeddings and not the actual correlation matrix rows to represent words? Why do we need to abstract--

RAMA RAMAKRISHNAN: Yeah, yeah, yeah. That's actually a good-- that's a good question. One immediate reason is that row is 500,000 vectors long. 500,000 long. So you want that compact, dense representation. The second thing is that thing is subject to all the counts of the Wikipedia corpus. It's not normalized. So you need to normalize it so that if you take any two rows and do dot product, you will get some number which is in a narrow range. Otherwise, things don't become comparable.

Now, both these objections can be handled. You can normalize. You can reduce the size of the corpus, and so on and so forth. And in fact, that used to be a very common way people used to do it before. But what they have discovered is that the way we learn embeddings now tends to be much more effective in practice.

AUDIENCE: So what we've done is-- what this process does is it creates this n dimensional incomprehensible matrix that captures, in essence, a summarized version of these relationships?

RAMA RAMAKRISHNAN: Right. Correct. A compact representation of relationships, which is not subject to the size of your vocabulary. So you find that 1,000 words today. Tomorrow, somebody comes up with a word called selfie, which didn't exist five years ago, and now your corpus has gotten a little bit more. So here, it's very compact, and it tends to have a much longer shelf life. Yeah.

All right, so let's see where we are. OK, so evaluate. 68-- 69%, almost. It was 63. We're going to 69. So clearly, here, training the whole thing-- including GloVe-- actually helps. And so that begs the question-- well, if training GloVe helps, maybe we should actually train the whole thing from scratch. Like, why the hell not? Why the heck not? I apologize.

So what we'll do is we'll actually create our own embeddings and just train them. And here, we don't have to worry about co-occurrence matrices and so on and so forth because we have a very specific objective. We want to be very accurate in predicting genre for these songs.

The people who had worked on GloVe, they didn't have any objective. They just wanted to create embeddings that were generally useful. Here, we want to be specifically useful for genre prediction. And so what we can do is we can actually train the whole thing ourselves.

We can actually give it-- we can actually put an embedding layer here. We just arbitrarily decided to choose 64 as the dimension, as opposed to 100. It'll run faster. And then it's the same thing. Global average pooling, activation, blah, blah, blah, blah, blah. And then you run it.

Let's see if it finishes in the next minute. And we'll see if it actually does better than the pre-trained embeddings or the pre-trained embeddings that have been further fine tuned. And I don't remember what I saw when I ran it yesterday. And while it's running, other questions? Yeah?

AUDIENCE: So my question is regarding embeddings. When you call embedding for a particular word, we indicate that we have certain amount of parameters. Let's say in this case, we have defined the predefined 100, so there'll be 100 parameters, and they'll be coefficient weights for each of them. So when we take a pre-trained model like the one we took from GloVe, so for each word, there would already be those number of parameters in terms of--

RAMA RAMAKRISHNAN: Right, yeah.

AUDIENCE: But then how do we redefine them as that we want only 100 or we want only 10 parameters...?

RAMA RAMAKRISHNAN: No, no. The GloVe thing actually gives you packaged-- it's prepackaged to be 100 long. I think they have 200 and 300 as well, if I recall. We just happen to use the one with 100.

AUDIENCE: The one that is available [INAUDIBLE]?

RAMA RAMAKRISHNAN: Yeah, yeah. And there are many available, we just get to pick and choose, and I happen to pick 100. Oh, OK. It's a bit slow, but it's actually looking promising. 9:55. Yeah?

AUDIENCE: So during the CNN models or even during our assignments, changing the filters gave us more depth and improvement in performance. So here, would I be right in concluding that it's actually training the embeddings, which is giving us more, assuming that epoch and batch changes are not going to change as much? So if I really want a genuine change in performance, we go to the level of retraining the embeddings? But--

RAMA RAMAKRISHNAN: Yeah. So what we saw was that using GloVe assist was OK. Using GloVe and then training them helped a lot. And now we are basically saying, well, what if we just abandon GloVe and train our own embeddings for our particular problem?

See, GloVe is a general purpose tool, so a general purpose tool is really good if you don't have a lot of data. That's a good starting point. But when you have a lot of data, you should always try to do your own thing and see if it's any better.

And in this case, I-- well, OK. I think it's-- Come on. It's 9:55. I thought I was going to enter any moment now. So-- right, let's look at the thing. OK, folks. So 72%. So you can actually retrain your own thing because there are 50,000 examples, and it gives you an even better thing. Thanks a lot. Have a good rest of the week.