[SQUEAKING]

[RUSTLING]

[CLICKING]

RAMA RAMAKRISHNAN: All right. So transformers, even though they were originally invented for machine translation, going from English to German and German to French and so on and so forth, they have turned out to be an incredibly effective deep neural network architecture for just, really, a vast array of domains. It has reached a point where if you're actually working on a particular problem, you almost reflexively will try a transformer first, because it's probably going to be pretty darn good. OK?

So they have just taken over everything. And obviously they have transformed translation, which is the original target-- Google search, information retrieval, completely transformed speech recognition, text to speech, even computer vision. Even the stuff that we learned with convolutional neural networks, now, there are transformers for computer vision problems that are actually quite good. Which is kind of shocking because they were not even designed for that.

And then reinforcement learning. And of course, all the crazy stuff that's going on with generative AI-- large language models, multimodal models-- everything-- everything-- runs on a transformer.

And then there are numerous special purpose systems. And I find these to be even more interesting. Like AlphaFold, the protein-folding AI-- runs on a transformer stack. And I could just list the examples one after the other. So it's just amazing. It's incredibly a flexible architecture. And I think we are lucky to be alive during a time when such a thing was invented. And I'm not getting paid to tell you any of this stuff. All right. It's just amazing.

So let's get going. We will use search-- or, more broadly, information retrieval, as the motivating use case. So these are all examples where people are typing in natural language queries or uttering natural language queries into a phone, and we need to make sense of what they want. And it's not, like, write me a limerick about deep learning, where there could be many possible right answers. It's more like, OK, tell me all the flights that are leaving from Boston to going to LaGuardia tomorrow morning between 8:00 and 9:00. Well, you better get it right. OK. Accuracy is a high bar.

Or how many customers abandon their shopping cart? Find all contracts that are up for renewal next month. Tell me all the customers who ended the phone call to the call center yesterday not entirely pleased with the transaction. The list goes on and on.

And so in particular, we will focus on this travel-related example today. Find me all flights from Boston to LaGuardia tomorrow morning. That kind of query.

And so in these sorts of use cases, a very common approach historically has been, well, we will take this natural language query and then we will convert it into a structured query. By that, I mean we will parse the query and we'll extract out key things in that query. Once we extract out those key things, we will reassemble it into a structured query, like a SQL query.

SQL is just one example of a possible structured query. There are many, many ways to structure queries. But SQL is familiar to lots of people, so I'm using that. So you take the SQL.

Once you have the SQL query, you're in a very comfortable structured land, in which case you just run the query through some database that you have, get the results back, format it nicely, and show it to the user. That's the flow.

So the question becomes, how do we automatically extract all the travel-related entities from this query? We want to be able to extract BOS, LGA, tomorrow morning. Flights. So on. These are all the travel-related entities we want to extract out. That's the problem.

And so we will use a really cool data set called the Airline Travel Information Systems Data Set-- and I'll explain the data set in just a bit. We'll use this as the basis for this example. And so the way we think about it is that we have a whole bunch of queries in this data set. And fortunately for us, the researchers who compiled this data set, they went through every one of these queries. And we have several thousands of them. They went through every one of those queries, and they manually tagged each word in the query with what kind of travel entity it is-- or none of them.

So for instance, so they call them slots. So they will take each word in the query and assign it to a slot, a particular kind of slot. And I'll explain what "slot" means in just a second. That's the basic idea.

So for example, if you have something like, I want to fly from-- OK? And this is a flights database, so you, can assume that everything is related to a flight flying. So if you have all these words-- 'I want to fly from"-- each of these words, these five words, gets mapped to something called the O-- which means other. It's the other slot. We don't really care about it. It's the other slot.

And then we come to Boston. Oh, Boston is very special, right? Because it's clearly a departure city. So we actually tag it. We assign it this label. Think of it as just like a classification problem-- a multi-class classification problem. So we assign it to B-fromloc.city_name. That is the label you assign it.

And then you go to at. You don't care about at. It's O-- other.

You come to 7:00 AM. And then, OK, that is depart time. So depart time. And then another depart time. And here you see there is a B and then there is an I, right?

So what we are saying here is that there could be entities who are described using more than one word, like 7:00 AM-- two tokens. And for that, we need to be able to figure out, OK. The second token is really-- is part of the first token. Together, they define the notion of a departure time. So what the B means that is that this is the word, this is the token in which we are beginning the idea of a departure time. And then "I" means we are in the middle of this description. B is for Beginning.

So you can see here, so there is a B here. And there is an I-- B for Beginning, I for Intermediate, or in the middle. And then "at," we don't care. 11, B-arrive_time. Morning, arrive_time.period.

So this is an example of how you can take a sentence and then manually label every word in the sentence with something that's relevant to your particular problem. And it turns out, these people, every word is classified into one of 123 possibilities. OK? So aircraft code, airline code, airline name, airport code. airport name, arrival date, relative name. Now, you get the idea.

They want a round trip versus a one way. The relative to today-- Because some would say tomorrow morning is relative to today. So you need absolute time, and you need a notion of relative time.

So they basically thought of every possibility with these researchers. And so every word in every one of these queries is assigned one of these 123 labels.

Any questions on the setup?

AUDIENCE: Did they have to contextualize what comes before relative Boston? So if someone says from Boston, so that there should be contextualization with the "from Boston."

RAMA RAMAKRISHNAN: So because they did it manually, they could just read it and figure it out, that's what they mean. Right? Boston is the departure city and not the arrival city.

AUDIENCE: So do they have to attach to Boston, which is some departure city as well as arrival city with the word "Boston"?

RAMA RAMAKRISHNAN: In the particular phrase, it's clear from that particular case in the context of it, as a human reading it, that Boston is a departure city. So it just only gets that tag in that sentence. In some other sentence, where people are coming into Boston, it will have a different tag.

AUDIENCE: I was wondering if my query, like the others, basically there is, for example, if my query was "give me flights from Boston at 7:00 AM," and flights from Denver at 11:00 AM.

RAMA RAMAKRISHNAN: You mean like a compound query?

AUDIENCE: Yeah?

RAMA RAMAKRISHNAN: So this one only takes single queries into account, because most people are like, give me a flight from here to there, or what is the cheapest thing from here to there? And we'll see examples of queries later on.

OK. All right. So that's the deal. So basically, what we have, this problem that we have here is really a word to slot-- word to slot-- multi-class classification problem. OK because if you look at that input, we want to be able to take that input. And a really good model will then give you this as the output, because this is what a human would have done. So that is our problem.

So the question is-- the key thing here is that each of the 18 words in this particular example must be assigned to one of 123 slot types-- each word. It's not like we take the entire query and classify the entire query into one of 123 possibilities. Every word in the query has to be classified. That is the wrinkle. OK?

So now, if we could run the query through a deep neural network, and generate 18 output nodes, it goes through some unspecified deep neural network. And when it comes out the other end, the output layer has 18 nodes. OK? Because that is the dimension of the output that we care about. 18 in, 18 out. 18 in, 18 out.

And then, for each one of those 18 nodes, maybe we could attach a 123-way softmax to each of those 18 outputs.

By the way, isn't it cool that we can just casually talk about sticking a 123-way softmax onto each one of 18 nodes? Folks, wake up. You're not easily impressed. I'm impressed by that. So, OK.

So here's the key thing, right. We want to generate an output that has the same length as the input. But the problem is, the inputs could be of different lengths as they come in. There could be short sentences, large sentences. We don't know. Yet we need to accommodate this range, this variable size of input that's coming in.

But the key thing is the output has to be the same thing as the input-- the same cardinality as the input. OK? That's a one big requirement.

In addition, we want to take the surrounding context of each word into account. To go to [INAUDIBLE] when you see the word "Boston," you can't conclude whether it's a departure city or arrival city. You have to look at what else is going on around it. Is there a "from," is there a "to." Things like that. To figure out how to tag it. So clearly, the context matters.

And then we clearly have to take the order of the words into account. Going from Boston to LaGuardia is very different than going from LaGuardia to Boston. So clearly the order matters. So the context matters and the order matters. And the output has to be the same length as the input.

So context matters. Just a few fun examples. Remember, from the last week, that the meaning of a word can change dramatically depending on the context. And we also saw that the standalone or uncontextual embeddings that we saw last week, like "glove," they don't take context into account because they give a single unique embedding vector to every word. And if a word ends up having lots of different meanings, that vector is some mushy average of all those meanings.

So the word "see," "I will see you soon." "I will see this project to its end." "I see what you mean." Very different meanings of the word "see."

This is my favorite-- bank. "I went to the bank to apply for a loan." "I'm banking on the job." I'm standing on the left bank," and so on.

It, the animal or-- this actually very-- it's a good one. "The animal didn't cross the street because it was too tired." "The animal didn't cross the street because it was too wide."

Can you imagine a deep neural network, looking at this word, "it," and trying to figure out what the heck does the word "it" mean? What is it referring to? Tricky, right?

And then, if you take the word "station"-- and I have this station example here because we're going to use it a bit more in the rest of the lecture-- the station could be a radio station, a train station, being stationed somewhere, the International Space Station. The list goes on. So clearly, order matters-- context matters.

And clearly order matters. You can come up with your own examples. Let's keep moving. OK?

So the transformer architecture is a very elegant architecture which checks these three boxes beautifully. It takes the context into account, order into account. And then whatever is produced out there is the same length as whatever is coming in.

And the reason it's called the transformer, it is because if 10 things come in, 10 things go out. But the 10 things that go out are a transformed version of the 10 things that came in. That's why it's called the transformer.

If 10 things came in, and like one thing goes out, well, sure, it's been transformed. But what is it? It's some weird thing. But when 10 comes in and 10 goes out, the 10 is preserved. Each one is getting transformed in an interesting way. That's why it's called a transformer.

So developed in 2017, this dramatic impact.

So by the way, the effect of transformer-- Google had spent a lot of research on machine translation and, obviously, search. And then, when the transformer is invented, they took a model called Bert, which we will see on Wednesday in detail. And then they introduced Bert into their search. And the results were dramatic. And from what I've read, apparently the impact of doing that was--

Typically, when you make an improvement to search, the improvement is very, very marginal, because it's already a very heavily optimized system. And then, when the transformer thing came along, there was actually a significant jump in search quality.

So for example-- and you can actually read this blog post which came out when they introduced Bert into search. It gives you a bit more detail. But here, so if you queried something like "Brazil traveler to USA needs a visa," you would think that it should give you information about how to get a visa if you're Brazilian and want to come to the US. But it turns out the first result was, how US citizens going to Brazil can get a visa.

So, clearly, it's not taking the order into account. But once they introduced it, boom, the first thing was the US Embassy in Brazil, and a page on how to get a visa. So the effect was dramatic.

And so this is a seminal paper. And it's actually worth reading the paper. And it's worth-- and this is the picture. This is, like, an iconic picture at this point, in the deep learning community. And we will actually understand this picture by the end of Wednesday.

But the funny thing is that when the researchers came up with it, they didn't realize, in some sense, like, what they had stumbled on, because they were really focused on machine translation. It's only the rest of the research community that took it and started applying to everything else and found it to be really, really effective.

So we're going to take each one of these things and figure out how to address them, and thereby build up the architecture. Any questions before I continue? Yeah.

AUDIENCE: Is there any benefits to discarding some of those unclassified O's before it goes out? So, rather than going like you have 18-word input. Discarding all the ones that don't actually matter and just doing, like, an 8-word output, for example?

RAMA RAMAKRISHNAN: Yeah, yeah, I think that's a totally fine way to think about it. Basically what you're saying is that can we have a two-stage model. The first stage model is like a O, non-O classifier, and the second stage model only goes after the non-O's. That's a totally fine way to do it. Yeah.

But as you can see, even if you go with just a simple one-stage model, if you use a transformer, you get fantastic accuracy. And we'll do the Colab in a bit.

All right. So let's take the first thing. How do you take the context of everything around the word into account? So let's say that this is the sentence we have "The train slowly left the station." For each of these words, we can calculate a standalone embedding-- say, something like "left." So I'm just depicting these standalone embeddings using these thingies here. Please appreciate them, because it took me a while to get them to do in PowerPoint. So these are w1 through w6. These are the vectors standing up Now let's say that-- so we can easily do that.

Now what we want to figure out is we want to focus on the word "station." And since "station" could mean very different things in different contexts, we want to figure out, how do we actually take "station's" embedding and contextualize it using all the other words that are going on in that sentence. Clearly, it's a train station. So we need to take the fact that there is a train involved to alter the embedding of the word "station." That's what taking context into account actually means.

So how can we modify "station's" embedding so that it incorporates all the other words? That's the question. So when you look at it this way, imagine, just for for a moment-- just for a moment-- that we--

Now some of the other words sentence don't matter. The word "the" probably doesn't matter. But some of the other words like "train," "slowly," "left" probably does matter. And suppose, just magically, we have been told all the other words in the sentence, this is how much weight you have to give to them. These don't give it any weight. Those give it a lot of weight. Suppose we are told that.

Or to put it another way, and this is the word that's heavily used in the literature-- someone tells you how much attention to pay to the other words, whether you have to pay it a lot of attention or very little attention. And this, how much attention to pay, is given in the form of a weight that you can use.

So if you look at it that way, from this notion of which word should I give a lot of weight to and very little weight to, in this example, intuitively, which words do you think should get the most weight, and which words do you think should get the least weight? Yeah.

AUDIENCE: "Train."

RAMA RAMAKRISHNAN: "Train." Correct.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: You can do one at a time. "Train." OK. Thank you. OK, others?

AUDIENCE: "Slowly."

RAMA RAMAKRISHNAN: "Slowly," right. So that also seems to have some bearing to it. What about words that don't really we don't think are going to help at all.

AUDIENCE: "The."

RAMA RAMAKRISHNAN: "The." Exactly. It probably doesn't do much here. Some context it actually might make a difference, but in this sentence maybe not. Intuitively.

So we should probably give a lot of weight to "train," maybe a little to "slowly" and "left," and hardly anything to "the." And so this intuition that we have can be written numerically as maybe we have a bunch of weights that add up to 1. Maybe something like this. So we are saying, the "train"-- 30% weightage. Maybe 8% weightage to "left." Maybe 12% weightage to "slowly."

And then, as you will see here, "station's" own embedding also plays a role, because we want to take its own standalone embedding and just move it slightly. Change it slightly. Which means that has to be the starting point. So it will get a lot of weight. You can't ignore itself, in other words. So we give it maybe 40% weight.

By the way, these numbers, I just made them up. Yeah.

AUDIENCE: I'm sorry. Just a quick question. So the weights are-- are they are they standalone to understand the context the entire sentence, or are they related to "stations" that we started off with, "stations."

RAMA RAMAKRISHNAN: These six numbers are only pertinent to "station."

AUDIENCE: Got it. Right.

RAMA RAMAKRISHNAN: And for each word we're going to do something similar.

AUDIENCE: Thank you.

RAMA RAMAKRISHNAN: Yeah.

AUDIENCE: At this point, does the model understand order? Because like, I'm just thinking of "left," because it gave it a very low-- a very low weight. But let's say "left" comes "slowly we left station." The station only happened to be higher.

RAMA RAMAKRISHNAN: Yeah, correct. So at this point, we not worrying about order. We are only really worrying about context. Later, we will take order into account.

AUDIENCE: How did the model know that "left" here is of lesser importance because it's a verb rather than an adjective.

RAMA RAMAKRISHNAN: It has to figure it out. We are just giving it a whole bunch of capabilities. How it manifests those capabilities is all going to emerge from training.

So, all right. So let's say we have something like this. So what we can do-- and we'll get to the all-important question of where do we get these numbers from in just a moment.

But suppose you had the numbers. How can we use these numbers to contextualize w6? What can we do? What is the simplest thing you can do? You have w6. You want to make it a new w6, which is now contextual. Is aware of what else is going on.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: It's working now, I think.

AUDIENCE: Take a weighted average.

RAMA RAMAKRISHNAN: We can take a weighted average. Exactly, exactly. So when you have a bunch of things and you have a bunch of weights, and we have to somehow modify one of those things with those weights, the simplest thing you can do is to take a weighted average. So that's exactly what we're going to do.

So we're going to take all these weights and just, like, move them up. Move them up. Don't even get me started on how long it took me to get this arrow run. I don't know about you folks. It is extremely painful to get the U-turn arrows to work in PowerPoint. Anyway, back to work.

So we just move these up here. So now we can do 0.05 times this vector plus 0.3 times that vector, and so on and so forth. And the result is just another vector, right? And that vector, folks, is the contextual embedding vector of "station." That was a standalone embedding. And now we did the-- we multiplied this by that, added them all up. And then you get a new vector.

And contextual embeddings have this bluish kind of color. And I'll maintain that color scheme as we go along. So that's it. That's it. That's the idea. Any questions? Yeah.

AUDIENCE: How did you come up with the original weights again? You just kind of guess?

RAMA RAMAKRISHNAN: No, these weights, I just hand type them in manually, just to make the point. And now I'm going to talk about how we are actually going to calculate them. OK. All right. Cool.

So now I'm going to-- OK, enough pictures. Let's switch to some math. So basically what I'm-- so let's write it a bit more formally. So we have these w1 through w6 which are the standalone embeddings. And then, for "station," we want to calculate w6 with a little hat on it, which is the contextual embedding.

And the way we do it is to say we calculate some weights for each of these words. So this weight, s1,6, means that the weight of the first word on the sixth word, which happens to be station. The weight of the second word on the sixth word, and so on and so forth. And so what we are saying is that w6 is just this weight times w1, this time w-- that's it. I have to inflict all these subscripts and all that because we need it.

All right. So that's it. That's what we have. Now let's talk about-- any questions on the mechanics of it before I get to-- OK, where do these weights come from? Yeah.

AUDIENCE: [INAUDIBLE] utilizing something like Google, for example. How does it understand the context of new words that comes to play? Is the process immediately through the training data the users put in, or what makes it--

RAMA RAMAKRISHNAN: Like a totally new word that didn't exist before?

AUDIENCE: A new word or a new context to a word that already exists?

RAMA RAMAKRISHNAN: No, I think that the context is supplied because the query coming into something like Google is a full sentence, and we only take that sentence and take only the sentence into account as the context for us. So the context is always present to us when we get the input.

But the other question you had of, OK, what if there is a brand-new word you've never seen before, for which there is not even a standalone embedding? What do you do then? So let's punt on that till Wednesday, because I have to talk about something called byte-pair encoding and stuff like that before I can answer that.

AUDIENCE: Really quickly-- does that immediately translate to their predictive search queries? Utilizing, like, for-- we have a new word, for example. Does that automatically get applied to the predictive search queries, like what we were saying, how to, and then just automatic.

RAMA RAMAKRISHNAN: Oh, you mean like the autocomplete? Autocomplete uses a slightly different mechanism. They had a very complicated non-transformer thing for a long time. I'm sure they have a transformer version now. But I'm not privy to how exactly they've done it. So I don't quite know how they do it. But what you're proposing is a reasonable way to think about it. Yeah.

AUDIENCE: My question is, you take the path, a particular number of parameters, as in, let's say, 10 of them. And then we have calculated the contextual version of w6. So this gets added as H2 or it is [INAUDIBLE].

RAMA RAMAKRISHNAN: Replaces.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: Yeah w6 becomes w6 hat.

AUDIENCE: And now we are extending that.

RAMA RAMAKRISHNAN: Correct.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: No, we lose it. And as you will see here, as it flows through the transformer, it's getting more and more and more contextualized. So it's a left to right flow.

All right. All right, great. So by the way, this thing that we did for "station," we will do it for each word in the sentence-- the same exact logic. Obviously, the weights are going to change. But what will happen is that w1 through W6 will become w1 hat through w6 hat.

The same exact logic is going to hold. OK. I just don't have the slides for it because it's a waste of time. The same exact logic is going to hold.

All right. Now switch gears and answer the all-important question of where are the weights going to come from. So the intuition here is really, really interesting and elegant. So, clearly, the weight of a word should be proportional to how related it is to the word "station." The word "train" clearly is very related to the word "station." The word "the," it's not clear how related it is. Probably not all that related.

So the relatedness matters to the weight. More related, higher the weight. Just intuitively.

So one way to quantify how related two words are is to take their standalone embeddings and calculate the dot product. So in case folks have forgotten about the dot product-- oops, that's not what I want. Here we go.

So let's say you have a vector. Let's say this is the vector for "train." This is the vector for "station." So the dot product of these two vectors-- and I'll write it as train, station equals, basically, the length of the vector for "train" times the length of the vector for "station" times the cosine of the angle between them. So how long is each vector, the product of the two, and then the angle between them.

Now, let's assume for simplicity that these links are roughly the same. They are just one unit length. Just roughly. So if you assume that, this thing, let's say, becomes 1, let's say. This thing becomes one.

So all the action is here. So all the action is here. So basically, the dot product of these two vectors is really the cosine of the angle between them.

So now the question is, if you have something like this, which are very close to each other, the cosine of a very small angle-- actually, the cosine of 0 is what? 1. So if the angle is really, really small, the cosine is going to be very close to 1, because 0 is 1. The cosine of 0 is 1. So this thing is going to be pretty close to 1.

If you have a cosine of two vectors that are like this, 90 degrees apart, what is the cosine? 0. They are orthogonal, right, which maps to the English, orthogonal. So the cosine of that is 0.

And then, if you have something like this where they're literally pointing in opposite direction, what is the cosine of that 180? Minus 1. So that's it.

So if these two vectors are very close to each other, the cosine of the angle between them is going to be very close to 1. If they are really kind of unrelated, it's going to be 0. If they're anti-related it's going to be minus 1. So that's how dot products capture this notion of closeness or relatedness. All right. Input.

So we can use the dot product of these embeddings to capture relatedness. And so-- OK, iPad done. So now what we do is, now that we know that dot products can be used, we can't use them as is, because we need to do one more thing to make them proper weights. And what I mean by proper weights is that we want the weights to be, first of all, non-negative. And we want them to add up to 1. That's what a weighted average actually is going to mean.

But these cosines could be negative, right? And so we need to now adjust them to make them proper so that every one of them is guaranteed to be non-negative, and they will add up to 1.

When was the last time you had to take a bunch of numbers which could be anything, and then somehow make sure that they are going to be positive, non-negative, and they add up to 1? When was the last time?

AUDIENCE: Softmax.

RAMA RAMAKRISHNAN: Yeah, softmax. Exactly. So we'll do the same trick. So what we'll simply do is we'll just exponentiate them. So this w1, w6, this angle bracket thing is the dot product. That's the notation I'm using. exp of that is just you exponentiate them, e raised to that. And once you exponentiate them, they all become non-negative. And then we just divide each by the sum of everything, so that the whole thing will become like a probability. It'll just add up to 1.

Makes sense? So that's how we take arbitrary numbers and make them proper weights. All right.

So to summarize, from embeddings to contextual embeddings, that's what we do. We take all the standalone embeddings. We calculate these weights using this formula. And then we just do the weighted average, and we arrive at the contextual embedding, and boom, done.

And so by choosing weights in this manner, the embedding of a word gets dragged closer to the embeddings of the other words in proportion to how related they are. So just imagine for a second, in this case, "station" obviously has many contexts. But let's assume for a second that it'll only have the train context and the radio-station context. In the current context, "train" is closely tied to "station," and therefore exerts a strong pull on it.

Now "radio" is also related to "station," but it doesn't appear in the word, in the sentence. So effectively it has a weight of 0. And that's the beauty of it.

And please do not ask me things like, I was listening to a great song on the radio station, and the train pulled out of the station. Transformers can deal with stuff like that. But yeah, but you get the idea-- the main idea.

So by moving "station" closer to "train" by paying more attention to "train," we are contextualizing the "station," the word, the embedding, to the context of trains, platforms, departures, tickets, and so on. It's like this portal into the whole train world. It's beautiful. The simple idea will get you there.

So this, folks, is called self-attention. What we just described is called self-attention. And it's the key building block of transformers. And so to summarize, standalone embeddings come in, contextual embeddings go out. Any questions? Yeah.

AUDIENCE: I'm still struggling a little bit with the intuition of the word. It's the weight of the word itself in its own contextual embedding. So the weight of "station" and the "station" embedding, how should I think about that.

RAMA RAMAKRISHNAN: Think of it--

AUDIENCE: Seem intuitively like it would be high for all contextual embeddings, but I assume that's not the case.

RAMA RAMAKRISHNAN: It'll be high. It'll be typically be a high number, because the cosine of the vector to itself is going to be very-- the cosine is going to be 1. So it's going to be pretty high.

But there's no guarantee it's going to be the highest, right? Because they are not actually-- the length doesn't have to be 1. They could be-- we try to keep them smallish, but they don't have to be.

So the way I would think of it is imagine that you take an average of everything else first, and then you average it with the old embedding. Effectively, it's the same as just calculating the different weights and averaging the whole thing together. Sure. Yeah.

AUDIENCE: A little confused [INAUDIBLE] you need the same number of inputs and outputs. But is this the reason why, because you need the contextual embedding, even if it's like a other word, and it's related definitely [INAUDIBLE].

RAMA RAMAKRISHNAN: Correct. Correct. Exactly. Exactly. And the other thing to remember is that by getting-- by keeping the inputs the size of the input cardinality intact as you move through the transformer stack, when you finally come out the other end, there is no loss of information. And at the very end, you can choose to aggregate, simplify, summarize, and so on and so forth. It preserves your optionality as long as possible.

AUDIENCE: Do you know how long [INAUDIBLE] going to take?

RAMA RAMAKRISHNAN: Yeah, so what we do is the sentence comes in. There is a whole notion of something called a context window-- or what is the maximum length of the sentence we'll handle. And that's a parameter you can set. And we'll come to that when you actually look at the Colab.

Was there a question in the middle? No. OK.

All right. So that is self-attention. And now, because that felt too easy, we're going to do a little tweak called multi-head attention. So this is the self-attention we just saw. What we can do is we can be like, you know what? Why can't we have more than this? Why can't we have more than one of these? So this is called an attention head-- self-attention head. We'll have multiple self-attention heads. And I'll come back to the top thing in a second.

But the question is, why should we have multiple self-attention heads? Because a particular attention head is going to pick up some patterns. The reason is because it will help us attend to the multiple patterns that may be present in a single sentence. So far, when I've been explaining, I've basically been looking at what the meaning of these words are, just the meaning of these words. But in any complicated sentence, you have to worry about grammar. You have to worry about tense. You have to worry about tone. You have to worry about facts versus opinions. There could be any number of complicated patterns that are sitting in a simple sentence.

Which means, well, there is just not one way to pay attention. There could be many ways of paying attention. There could be many needs to pay attention. Which means that let's have many of these attention heads, and each one could be learning something else.

It's exactly like having lots of filters in a convolutional network. One filter might learn a line, another filter might learn a curve, and so on and so forth. And we don't want to decide a priori, oh, you're going to learn a line, right?

Similarly, here, we're not telling any of these things on what you have to learn. They just have to learn based on the training process.

So what we do is-- so actually, this is an example where this, from the original transformer paper, where this sentence is "The lawyer will--" sorry. "The law will never be perfect, but its application should be just. This is what we are missing, in my opinion." A complicated sentence, right?

So the first one, attention head-- actually, this is the pattern of things. So for example, the word "perfect" here. the contextual embedding of the word "perfect" draws upon heavily from the word "law" in this example. If you look at another attention head, the contextual embedding for the word "perfect" is actually drawing heavily from just perfect and nothing else. And if you look at other words, the patterns are subtly different of what it's paying attention to.

So these are two different attention heads. And they are learning different kinds of attentions. In reality, trying to make sense of why they pay attention to the way they do, it's usually quite difficult to figure that out. You can't actually interpret it. But when you have lots of attention heads, the performance on the task that you care about gets really much better. And then you say, OK, I can use that. Yeah.

AUDIENCE: Not to oversimplify anything. Is this -- is the idea behind this [INAUDIBLE]?

RAMA RAMAKRISHNAN: Exactly. Same logic. Same logic. Yeah.

AUDIENCE: [INAUDIBLE], with that filter, we had designed a particular filter for a particular task in terms of 1's and 0's on that line. But here, like the attention head, how do we differentiate-- how do we create an instance for each different kind of scenario or different kind of parameters we want to all kind of different kind of pattern [INAUDIBLE].

RAMA RAMAKRISHNAN: Actually, in the convolutional case, the 1's and 0's I had were just example numbers to show that that particular filter could detect a vertical line or a horizontal line. You will recall that when we actually train a convolution network, we actually don't specify the numbers. We start with the random initialized weights, and then we let backpropagation figure it out. Similarly, here, we don't decide any of these things. We just let backprop figure it out.

And now the question of what are the weights that are actually going to be learned, we'll come to that in a bit. Yeah.

AUDIENCE: I was wondering, how come we have different attention heads, even though it seems like they're only function of the dot product, and we have the same dot product for the same embeddings.

RAMA RAMAKRISHNAN: Great question. Great question. And I literally have a note in my slide saying if a student asks this good question, tell them to wait till Wednesday. So, great question. And we'll come back to that on Wednesday and spend a fair amount of time on it.

So the point that's being made here is that-- oops. When we look at self-attention, the embeddings came in, and we did all these dot products, and the contextual things popped out the other end. Note that inside the self-attention box there are no parameters. There are no parameters.

So the question that is being raised here is that, so what are we learning, really, if there is nothing inside to be learned? If there are no parameters, no coefficients, what are we learning? That's the question. And by extension, if we have two of these, and neither of them is learning anything, what's the point? Sadly, you have to wait till Wednesday. But we have a great answer to the question, so it'll be worth it. And if you can't stand the suspense, read the book.

All right, so that's why we need multiple heads. And now to come back to this, so what we do is, it goes through this head, and you get these w1's. And it goes through here, and we get another set of w's. Then, what we do at the very end is we concatenate them. OK. We concatenate them. And we do a projection. And this is what I mean by that.

So we have-- this is one self-attention head-- self-attention 1. This is self-attention 2. And let's say that. w1 hat comes out. And I'm just going to call it Z1 for the same thing so that there's no name clash. And w2, w6, all of them are coming, right? Let's focus on w1 and z1.

w1 and z1 are both contextual embeddings for the same word, for the first word, word 1. And so what we do is, let's say this w1, let's say this vector is like this. And let's say that this vector is like this. What I mean when I say concatenated here is we literally take this word here, this embedding here. Then we take this thing here. And we just make it a long vector. We concatenate it.

But now, this vector has become twice as long, right? But remember, we always want to preserve the number of inputs we have and the lengths of these vectors everywhere as we go along. So what we do is, at this point, we run it through a single dense layer, which will take this thing and make it back into the same small shape as before. So this is a dense layer. That's it. So this vector comes in, and it gets compressed back to the original shape that came out of here.

So you could have 20 of these attention heads, and the concatenating will be 20 times longer. And then you just project, boom. One dense layer comes back to the original shape. So that is the projection step. And that's what I mean here when I say concatenate and project.

So at this point, what we have is things come in. We contextualize them using these different attention heads. And when they come out of the attention heads, we take them all. We just concatenate them and then compress them back to the same original starting shape. If these vectors are 100 units long or 100 dimension long, whatever comes out is 100 still. And to preserving this size as we go along is very important for reasons that will become apparent a bit later. So that is the multi-attention thing.

Now, a final tweak for today is that we will inject some non-linearity with some dense layers, dense ReLU layers at the very end. So we went through a bunch of attention heads. We came up with a bunch of contextual embeddings now. So at this point so far, there are no-- since there are no parameters inside these boxes, and there are some parameters here, we need to do some non-linearity. So far, there has been nothing that's nonlinear so far. So here we actually send it through one or more ReLUs. Typically, they just use one ReLU.

And what I mean by that-- oops, sorry. So this is what we had here. And then we take it in and then run it through-- actually, we typically run it through a ReLU. This is a nice ReLU.

And then the rule of thumb, as you will see, if, let's say, this vector is, say, 100 dimensions long, they typically will choose a ReLU which is about 400 wide. And then it just gets projected out again back to 100.

So this is just a simple-- the input comes in, goes through a single hidden layer with four times as many as here. And then it project another dense layer to 100 again. And since there are ReLUs here, we have injected some non-linearity into the processing.

Now, a lot of this stuff, when it came out, felt very ad hoc, right? It didn't come from some deep theoretical motivations, but people had strong intuitions as to why these things were helpful. And as it turns out, since the transformer came out, people have tried to optimize every aspect of this thing. It's actually pretty difficult to beat the starting architecture. Improvements have been made, but it's actually a very robust architecture.

So that's what's going on here. And then, when we come out of this thing, this is what we have-- the story so far. We start with random standalone embeddings. These could be glove embeddings. It could be random weights. It doesn't matter. It goes through a bunch of self-attention heads. We concatenate it when it comes out the other end, concatenate it when it comes out the other end. And then we project it back to the same size as before. Then we run it through a ReLU followed by a linear layer. And we get these things again.

So in this whole process, if six things came in, six things will come out. And if those six things that came in were standalone embedding vectors of 100 dimensions, what comes out is also 100 dimensions. So in that sense, you could think of this whole thing as a black box in which whatever you send in, the same number of things will come out of the same length. The numbers will be different, because they've now been heavily contextualized. The numbers are much smarter, in other words.

So far, what we have seen is that we have satisfied two of the three requirements. We have taken the context of each word into account by using these dot products in the self-attention layer, and we can generate an output that is the same length as the input. But we have ignored the fact that-- we have ignored word order completely. Because whether I had said "the train slowly left the station," or I had said that, "The station slowly left the train," this thing won't know the difference, because dot products function on sets, not on sequences. They function on sets.

You should convince yourself of this. Regardless of the order, the dot reduction doesn't change anything, because we are doing every pair.

So the question is, how do we take the order of the words into account? As I was saying, we can scramble the order of the words in a sentence, and we'll get the exact same contextual embeddings at the end.

So by the way, if you're working on a problem in which the order doesn't matter, then you could stop right now and use the transformer. And there are many problems that are actually in that category, where the order doesn't matter.

So if you take traditional structured data, or tabular data-- blood pressure, cholesterol level, boom, boom boom. Does it predict heart disease? Well, there is no order in that thing. You can use a transformer as is, without doing anything more. So transformers work for both sets and sequences where order matters.

OK. So the fix for this is something called the positional encoding. So what we do is very simple. By the way, there are many things that have been invented to tell transformers-- to give transformers some information about the order of each of the things that are coming in. I'm going to go with something called the-- simplest possible way, which actually works pretty well in practice.

So what we do is, for each position, each possible position in the input, starting from the first position all the way through the last position, we imagine that that position itself is a categorical variable. If a sentence can only be 30 words long, let's say, we say that, hey, the position of each word is a number between 0 and 29. And so we can just think of it as a categorical variable. And because a categorical variable, we could just imagine an embedding for that, for each potential value. So it will become clear in just a moment because I have a numerical example.

And so what we do is we will just take that standalone embedding, and then we'll take this position embedding, which represents the position of the word in the sentence. And we'll just add them up. Yeah.

AUDIENCE: So if, in the initial sentence, it says, I have made a mistake. So I just write it as, "The train slowly a station." So which means my output is actually going to be wrong.

RAMA RAMAKRISHNAN: Yes. Now the transformers are, since they're trained on lots of data, they will be quite robust to these things. But strictly, arithmetically speaking, correct. Yes.

So let's look at an example. Let's assume that your standalone embeddings-- right? This is your vocabulary. "unknown," "cat," "mat," "I," "sit," "love, "the," "you," "on." That's it. That's our vocabulary.

And for this vocabulary, we have these standalone embeddings. And just for argument, let's assume these embeddings are only 2 long. The dimension of these embeddings is 2.

If you recall, the glove embeddings we used last week, I think they were about 100 long. And the ones we're using in the homework are even longer than that. But here, we are assuming they are only 2 long. So the embedding for "cat" is 0.5, 7.1

All right. Now let's assume that we can have, at most, 10 words in any sentence that's coming in. And obviously, a particular word could be in position 0 all the way through position 9. And we will learn embeddings for each of these positions. And these embeddings are also 2 long-- 2 units long, dimension 2.

Now where will these embeddings come from? What's the answer to that question? What is the answer to the general question of where will these weights come from? We will learn it with backprop. We will start initially with random numbers, and then we'll get them, make them better and better as over the course of training.

So what we do is we have these two tables of embeddings-- the standalone embedding for the word and the position embedding. And then, we literally add them up. So for example, let's say the word, the sentence that came in is "Cat sat mat." That's a sentence. It's got three words-- "cat," "sat," "mat."

So what we do is we say, well, the embedding for "cat" is this thing here, 0.571. So I write it here, 0.5, 7.1. "Cat" happens to be the zeroth position of the word. So I grab the embedding for 0, which is 1.33.9. I stick it there. And then I literally add them up, 0.5 plus 1.3, 1.8, 11.0. That's it.

So now the positional encoded embedding for the word "cat" is 1.8, 11.0-- not 0.5, 7.1. So if "cat" happens to show up in another part of the sentence-- let's say instead of "Cat sat mat," we had "Mat sat cat." Now "cat" is now the third position, which is 0, 1, and 2, which means its embedding doesn't change. It's just embedding for "cat." But now, instead of picking 0, we'll pick this one-- 0.681-- and put that here and add them up instead.

So this is the idea of the positional encoding. This is how we inject position knowledge into the transformer. Yes.

AUDIENCE: The position embedding would be different for each sentence? Like, how do you--

RAMA RAMAKRISHNAN: No, this is just one table, which tells you what the position is. So it says for a word that appears in the seventh position in any input sentence that you're feeding in, this is the embedding that you need to use for that position.

AUDIENCE: If a word appears twice in the same sentence, how do we [INAUDIBLE]?

RAMA RAMAKRISHNAN: Great question. So let's say, just for argument, let's say the sentence was "Cat cat cat." So for each one of those cat-- for "Cat cat cat," this embedding will be the same-- 0.5, 7.1-- because that happens to be just the embedding for "cat" regardless of position.

But then the first "cat," for the first "cat," we will use 1.3, 3.9 as the addition. For the second "cat," we'll use 6.3, 3.7, and the third "cat," we'll use 0.6, 8.1. So only the things that are adding, the positional encoding will change, the positional embedding. So the resulting sum is going to be different for each of these three words, even though they're exactly the same word.

AUDIENCE: Is that position embedding table specific to that standalone embedding table? If you were to add or remove some words from the standalone?

RAMA RAMAKRISHNAN: It's independent. Independent. It only depends on your assumption about how long the sentences can be. That's it. It doesn't really care about what words are coming in. That's a whole different thing. So these are two independent tables that have just learned as part of this process.

So yeah, I have the same thing for "sat" and "mat," "sat" and "mat." That's what we have. So just make sure you understand these two slides to really make sure the mechanics are clear. Yeah.

AUDIENCE: How do you filter out for filler words? For example, if you're taking NLP output for transcription, and you were trying to run a transformer on it, and you have a lot of "ums" and "likes" that are disproportionately large, and have these random assignments or really deep embeddings. Are there ways to filter out a certain noise?

RAMA RAMAKRISHNAN: Typically, what they do is, as we will-- we'll talk about this thing called byte-pair encoding, in which we take individual characters, fragments of words, and words into account as tokens. So when you hear stuff like "um," "uh," and so on, it gets mapped to these small tokens. And then we treat them as just any other token.

AUDIENCE: Yeah, is aggregation, like, a simple sum or [INAUDIBLE], the actual semantic meaning of the word in the standalone embedding might be more important than its relative position in the sentence?

RAMA RAMAKRISHNAN: It could be. We just don't know a priori whether it's going to be important or not for any particular sentence. When we train the transformer with a lot of textual data, it will just figure out the right values for these things so that on average, the accuracy is as high as possible.

So in many of these things, there's always a tension between our human intuition as to how it should work, and whether you should just throw it into the meat grinder of backprop and see what happens. And so here, as it turns out, you can just throw it into backprop. It'll actually do a pretty good job. Yeah.

AUDIENCE: For the positional encoding, we would just be using the sum vector. We wouldn't be using this 2 by 3 matrix that you have on the far right, right?

RAMA RAMAKRISHNAN: Oh, yeah. This is just for demonstration. Basically, this is the thing that will actually go into the transformer. Correct. Yeah. That was just me being overly verbose in the slides. Yeah.

AUDIENCE: [INAUDIBLE] sentence the input at this point, are we still parsing out punctuation, or do we have a multi-sentence input? Is there positional embedding vector for each of the sentences?

RAMA RAMAKRISHNAN: Yeah. So here, basically, the starting point is tokens, right? And in our example, because we're working with the idea of simple standardization and stripping and things like that-- I'm just showing actual words-- if you go to something like GPT-4, since it uses a different tokenization scheme, each token might be part of a word. It might be an individual character. It might be a punctuation mark. In fact, the GPT family doesn't support punctuation, which is why when you ask a question, it comes back with intact punctuation in its response. And so we'll revisit this when we look at BP byte-pair encoding later on.

But the key thing to remember is that all the stuff you're talking about starts from the notion of a token. As to how you define a token, given a bunch of text, that's the tokenizer's job. And we just assumed a simple tokenizer for the time being. So at this point, folks, we have satisfied all the requirements. We have taken the surrounding context of each word, we taken the order, and so on and so forth, because what's coming in here is the positional embeddings. And it runs through the whole transformer stack.

So this is called a transformer encoder. This is the transformer encoder. And you can see here, this is the original picture from the paper. It's an iconic picture at this point. So it says here, these are the inputs. This is like "The cat sat on the Mat " It comes in here, gets transferred to-- transformed into embeddings, standalone embeddings. And then, based on the position of each word, we add-- that is why you see a plus sign here. We add the positional embedding to that, and the resulting thing goes into this transformer block.

And here, we go through multi-head attention. And things come out the other end. Then, there is this thing called add and norm, which we'll revisit on Wednesday. And then it goes through a feed forward network, another add and norm, which we'll revisit on Wednesday. And then it comes out the other end. That's it. That's the transformer encoder.

And so if you look at this, just to point out a couple of things, the input embeddings can be random weights, or it could be pretrained embeddings. We add in a position-dependent embedding to represent the position of each word in the sentence. That's the plus. Then we pass it through multi-headed attention to get a contextual representation. Then, finally, we pass all this through a simple-- typically, it's a two-layer network, a one hidden layer with ReLUs and then a linear layer after that, and boom. And then we do it. This is the encoder.

And here is perhaps the most important point to keep in mind. Because we have taken inordinate care to make sure that the things that are coming in and the things that are going out have the same size, both in terms of the number of tokens as well as the length of each vector, we can then stack them up like pancakes. We can have lots of transformer stacks, one on top of each other. Because it's the perfect API. It's the simplest possible API. The same thing comes in, same thing goes out, in terms of size.

So you can have a transformer encoder, another one top, boom, boom, boom, boom, boom, one after the other. GPT-3 has 96 transformer stacks. And like in all things deep-learning related, the more layers you have, the more complicated things we can do with it, as long as you have enough data to keep the model happy so it doesn't overfit. OK? All right.

So what we haven't covered, which we'll cover on Wednesday, is the question that he had posed, about how, since there are no parameters inside the self-attention block, what are we actually learning? And then there is these things called residual connections and layer normalization. We'll talk about all those things on Wednesday. Those are all refinements to the idea.

So all right. 9:39. Let's apply the transformer encoder to an actual problem. Any questions? Yeah.

AUDIENCE: My question, regarding, like, you said you could have multiple transformers. What is the difference with having multiple self-attention heads, and rather than having multiple [INAUDIBLE]?

RAMA RAMAKRISHNAN: When I say a transformer block, within the block, there could be multiple heads.

AUDIENCE: So if you increase accuracy, as you say, by [INAUDIBLE] this this way

RAMA RAMAKRISHNAN: Yeah, you can have a lot of attention heads. And that's totally fine. And typically, I forget how many GPT-3 and 4 have. They have a whole bunch of them. But you can-- so you can go wide and you can go deep. Both are done in practice.

But the thing is, the one thing you have to remember is that if you go wide, you have a lot of attention heads, then, given the particular input that's coming into that block, it will learn different patterns from it. Well, if you stack them all up, it's going to learn different ways to contextualize the things that are coming in. It operates at higher levels of abstraction.

So the analogy would be that, the seventh layer of a convolutional net, take the sixth layer's output and say, oh, I'm seeing a lot of edges here. I'm going to take an edge like this, two circles like that, and call it a face. So it will operate at a high level of abstraction.

All right. Let's go to the Colab. So what we're going to do is we're going to take the transformer that we just learned about, and we're going to apply it to solve the travel slot problem. OK? All right.

So OK. So we'll start with the usual preliminaries. And then we have taken the eighth test data set I talked about, and we have stuck them in raw box for easy consumption. It's here.

So if you look at to the top view, you can see here, for example, I want to fly from Boston, 8:38 AM. And then this is the output. And the slot filling is the output. And so, as it turns out, here there is-- these people also gave it-- they took the whole query and gave it an intent as to is it-- it's a flight query. It's a something else query and so on, which we're not going to-- are you kidding me?

I want to fly from Boston at 8:30 AM and arrive in Denver at 11:00 in the morning. What kind of ground transportation is available in Denver? What's the airport at Orlando? How much does the limo service cost within Pittsburgh? And so on and so forth. So you get the idea. It's a very wide range of queries that are in this data set.

So let's just ignore that for a sec. So what we're now going to do is we're going to take only this column, right, the query column. That's going to be our input text. And then the slot-filling column is going to be our dependent variable, the output. So we'll just gather them all up here. Let it run. We'll do it for the training data and the test data.

And so what we have done is that we have taken the transformer-related code in Keras, and we have packaged it into a little HODL library for easy consumption. And so that thing is here. You can download it. Calling it a library is overstating it. We literally just created a bunch of code and stuck it in a file.

And so what we'll do is from HODL, we'll import the transformer encoder. And we will import this positional embedding layer, because what we are going to do is we are going to take the input, do the positional encoding business, and then send it into the transformer. OK?

But first, let's vectorize the input queries that are coming in. So we'll define a thing here. This is a-- max query length is not defined. That's what happens when you don't run everything. All right.

So now we have this thing here. So turns out that there are 888 tokens. 888 words in the input queries that we have in the data. So take a look at the first few. And you can see here, there is UNK. And because the output mode here is-- you just want integers to come out, not multi-hot encoding or anything, because we're going to take those integers and then do embeddings from them.

It will reserve this empty string as the pad token. This should be familiar from last week. And then the UNK, for unknown tokens, and then 'to' 'from' 'flights' These are some of the most frequent. Turns out Boston is actually the most frequent. I don't know what's up with that, but it is what it is. Then we'll do the same vectorization to the train and test data sets.

Now we need to do STIE for the output side of the problem, because the slots, the dependent variable here, remember, are all sentences as well, with the B, O, things like that, right? So we need to vectorize those, or we need to do STIE on them. So let's take a look at some of these slots. And you can see here, all this stuff is going on.

Note-- so now, here is an example where you have to be very careful when you do the standardization. Typically, standardization, you will remove punctuation, and do things like that in lowercase. But here, these things have a specific meaning. We can't just go in there and remove the period and the underscore and then make the B into lowercase B and stuff like that. That will just harm it, right? We need to be able to preserve the nomenclature of the output in terms of all those tags.

So we don't want the standardization to do all those out. So what we do is we say standardization, none. Look at that. We tell Keras, do not standardize this. Do not do your usual thing.

So we do that for the output side. And then let's look at the vocabulary. Yeah, so this sounds pretty good. These are all the things that we would expect to see. These are the distinct tokens in the output strings. All right.

OK. We get it. So we have 125 of them. In the lecture, I said there are 123 slots. Possible slots. Why is it 125 here?

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: Yes, uncompared. Correct.

OK, now we'll set up a transformer encoder. Oh, wait, wait, wait. I forgot about doing this. My bad. All right.

I just thought, when I saw this slide, that we should go to the Colab without giving you a bit more background. No problem.

So the way we are going to model this problem is that we're going to have something like this-- fly from Boston to Denver. That's the input that's coming in. And that is the correct answer. 0, 0, some B, something or the other-- I mean, O, and then something else. That's the correct answer. That's the input, and that is the right answer.

So what we'll do is we will create these positional input embeddings like we have discussed before. We will run it through a transformer. It gives us contextual embeddings. So if we send five in, it's going to send us five out, except the color is now blue.

And then what we do is we will run it through a ReLU. OK? We'll run it through a ReLU. We will still have five vectors here. Five vectors will come in. And then, for each of the things that comes in, we will stick a 123-way softmax. For each thing that comes out, we'll have a 123-way softmax. And that's the classification problem we're going to solve.

So the weights in all these layers will get optimized by backprop. All these weights are going to get optimized. Yeah.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: Sorry?

AUDIENCE: What about the positional embeddings imported in the Colab [INAUDIBLE]?

RAMA RAMAKRISHNAN: No, that's the layer. The weights in the layer will still need to be learned. It's like the text vectorization layer. It's a bunch of code. And then you actually run it on a particular corpus to adapt it and fill a vocabulary out of it. So it's like an empty shell that needs to get populated.

OK, so with the weights in, all these things are going to get updated when we train the model by backprop. And that's it. That's the setup.

Does this make sense, before I switch back to the Colab? In particular, does this make sense, this part of it? Bunch of things come out. And then, for each one of those things, we need to figure out a classification, a 123-way classification. And that's where we stick a softmax in every one of those output nodes. Yeah.

AUDIENCE: [INAUDIBLE] for the yellow vector layer, is there a reason that they're, like, 3-dimensional?

RAMA RAMAKRISHNAN: Oh, oh, I see.

AUDIENCE: Not three. But why it's changed from three to four?

RAMA RAMAKRISHNAN: That's a good--

AUDIENCE: Or is it just kind of random, like [INAUDIBLE].?

RAMA RAMAKRISHNAN: It could be whatever. Or to put it another way, it is your choice as the user, as the modeler. Correct.

The thing is, at this point, with the blue stuff, the transformers basically say, my job is done. It has given you these valuable contextual embeddings at some high level of abstraction. What you do with it depends on your particular problem. And so the best practice would be to take it. And then maybe if these embeddings are really long, maybe make them a little smaller using a ReLU. And using a ReLU is always a good idea, because when in doubt, throw in a bit of non-linearity.

And then, once you are done with that, well, at this point you need to actually classify it. So you stick an output softmax on it. So that's what we have.

All right, back to this picture. So what we're going to do is we also get to decide how long are these embedding vectors. How long. Because here, we're not going to use glove embeddings. We're just going to learn everything from scratch. We're going to learn everything from scratch.

And we can decide how long these embedding vectors are. So these embedding vectors, I'm going to decide. I have decided that I want them to be 512 long. I want these actually to be 512 long. So that's what I have here-- 512.

And then, inside the transformer, remember, when we concatenate everything, and then we have something, we run it through a final ReLU layer. How big should that layer be? That is what, here, what I mean by dense_dim. I want it to be 64.

And then I, for fun, I'm going to use five attention heads, because why not? And then, in the final thing here, to go to Alina's question here, these things are all 512 long, as I mentioned earlier. These are all 512. But this thing here, I'm going to make it just 128. That's what I mean by units here. And so if you look at the actual model, OK, whatever comes in has a max query length of, I think, 30, if I recall. Actually, let's just make sure of that. What did I assume? 30. Correct. Max query length 30. So each sentence is 30.

So if a sentence has 35 words in it, what's going to happen?

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: The last 5 will get chopped, truncated. If it comes in at 22, we're going to pad it with eight more tokens, with a pad token. That's how we make sure everything gets to 30.

All right. So we come back here. So the input is still sentences which are 30 long, tokens which are 30 long. And then we run it through a positional embedding layer. OK? This positional embedding layer has the actual embedding for each word in that table, and it has the positional table, the positional embedding table.

So just to be clear, this positional embedding layer is basically-- it's basically this. So this table and this table together are packaged up into the positional encoding layer. But they are totally distinct tables. They just happen to be packaged up.

So this is what we have here. And then we get a nice positional embedding out. And then, boom, we run it through the transformer. And this transformer encoder object, we have to tell it, obviously, hey, this is the embedding dimension that's going to come out. This is the dense dimension you're going to use in that final feed forward layer inside each attention block. And this is the number of attention heads I want you to use. That's it. It's very-- right? Only three things have to be specified.

And then, whatever comes out of the transformer encoder are these blue vectors. And then we are back into good old traditional DNN stuff, where we take this thing, run it through a ReLU with 128 units. We add a dropout. And then we run it through a dense layer, which the vocab size here is 125, which is the 125-way softmax. Activation softmax. Connect up everything into model input and output. And boom, that's the whole model.

So that's what we have here. Now, this, after Wednesday's class, for extra credit and for your personal edification, try to work through this thing to come up with this number-- 53 million-- sorry, 5.3 million. And see if it matches this number here. It should match.

Hand calculate the number of parameters inside the transformer. For fame and fortune. That's an optional thing. So do it after Wednesday's class, not right now. And I have actually listed the exact math that goes into it here.

All right. So, by the way, you can peek into any layer's weights using its weight attribute. This is the embedding, the positional embedding thing we had. So we can click it. And you can see here, it has two tables. There is the first table, which is just the embedding table, which says there are 888 tokens in my vocabulary, and each of those tokens is an embedding vector, which is 512 long. That is the first table here.

And then it has the second object, which is the positional embedding. And it says here, well, my sentences can be 30 long. And for each position of the 30-long sentence, I will have a 512 embedding. Both these tables, as I mentioned earlier, are packaged up inside. And you can actually see what the weights are before you do any training.

So all right. So I'm going to stop here, because the model, it's going to take a few minutes to run. And we are already at 9:45. So we will continue the journey on Wednesday. If some of it is not super clear, don't worry about it. It'll become much clearer on Wednesday.

All right. All right, folks, have a good couple of days. I'll see you on Wednesday.