MITOCW | 15.773 SP24 Lecture 8 Version 2.mp4

[SQUEAKING] [RUSTLING] [CLICKING]

RAMA RAMAKRISHNAN: OK, so we'll continue with transformers today, part two. We are going to do the second pass. This is going to be a deeper pass through the transformer stack. And I think maybe the next 30 minutes, it's potentially the most demanding 30 minutes of the entire course, OK? With that motivational speech, let's get going.

So a quick review. Why do we want transformers? Because we want an architecture that can generate output that has the same length as the input, same length. There it is. Number two, we want to take the context into account, and we want to take the order into account. And as you saw last time, the transformer architecture delivers on those three requirements.

And so just a quick review. If you have a phrase like, "the train left the station," we have all these little arrows which stand for the standalone or uncontextual embeddings. And then-- sometimes this works, so I'm going to put it close to me here. So if you're here, if we start with either standalone embeddings, i.e. uncontextual embeddings, which have been pre-trained or random, doesn't really matter, if you look at the Colab we did the other day, we actually just started with random weights for the embeddings, and then we add positional embeddings to them.

And so each embedding, each word here, we take its standalone, we take its positional embedding, we literally just add them up element by element. Then we get a total embedding, and that's called the positional embedding of each word. And then that's what we have, positional input embeddings. So this whole thing goes into this transformer encoder stack, and what pops out the other end is contextual embeddings So that's the overall flow.

Now we applied this, the transformer stack, to the word-to-slot classification problem, where we basically took every incoming natural language query that comes in, we calculate its positional embeddings, and then we run it through the transformer stack, and then we get contextual embeddings. And then at this point, since each word that comes out, each embedding that comes out needs to be classified into one of 125 possibilities, we run it through a ReLU, when we attach a softmax to each embedding. This is basically what we did last class. So this is the transformer encoder. Actually, any questions on this before I continue on?

AUDIENCE: I was wondering why, when, how do you decide where to add more self-attention and where you add the transformer layers? You mentioned that ChatGPT has 96 of them.

RAMA RAMAKRISHNAN: So GPT3 has 96 transformer blocks. Each one is a block. So I think the question goes to, do you add more attention heads within a single block or do you add lots of blocks? And both are good things to do. What increasing the number of attention heads in a block does for you, it allows you to pick up more patterns at that level of abstraction.

But if you add more blocks, much like later convolutional filters can build on earlier convolutional filters, you're going up the levels of abstraction. So to go to vision, for instance, you have the notion of lines and so on in the beginning, and then you have a notion of edges, which are two lines. Then you have nose, eyes, face, and so on and so forth. So both are worth doing.

So typically that's what you typically find, that people typically have maybe a dozen heads or five, six, a dozen heads. We'll see examples of how many heads in a couple of architectures later on today. And the more you go up, the more capable the model becomes as long as you have enough data to train it well. So the perennial question of, do we have enough data to train this large model-- because if you don't have enough data, we might run into overfitting problems and so on. That's always the tradeoff.

So here, I just want to quickly switch to the Colab because we didn't have a chance to finish it. I'm not going to run it because it's going to take some time. So where we left off last time-- so here we basically took this architecture that we just saw on the slide, and then we essentially wrote it as a Keras model. And I went through this model in the last class, so I'm not going to go through it all over again.

What we did not do last class was to actually run it. And so if you actually run it, you can just run it for 10 epochs, just like we normally do. Give it data, give it a bunch of epochs, choose a particular batch size. I just arbitrarily chose 64. You run it for 10 epochs, and then you evaluate it on the test set. You get a 99% accuracy on this problem. One transformer stack, that's it-- one block, rather, one block. That's it.

And of course, here, there is a little trickiness going on here because a naive model can literally say every word that comes in is "other," O. And since the Os are the majority of the words, it's not going to do badly. It's like having a classification problem in which one class is very predominant. So the naive way to actually do well is to just say every time something comes in, oh, it's that majority class.

The same thing happens. But if you don't adjust for that, it turns out that the accuracy on the non-O slots, which is really what you care about, is actually 93%, which is actually pretty good. And then I had some examples of lots of fun queries you can do, including queries where I try to break stuff like cheapest flight to fly from MIT to Mars and see what happens, things like that. So have fun with it.

All right, back to PowerPoint. So this is what we had. Now what we're going to do in today's class, we are actually going to take the encoder we built last time and introduce three new complications into it. And when we finish introducing these three complications, we will actually have the actual transformer that was invented in the 2017 paper. The first tweak is the hardest tweak, so we'll slowly work our way to it.

So the thing to remember is, let's review self-attention. What is self-attention? You have a bunch of words, and we further said that for any particular word, like "station," we want to take its positional embedding and then make it contextual. And the way we do that is by taking each word's embedding and then calculating these dot products between all the other words. And then since these dot products can be positive or negative, we want to make them all positive and normalize them so that they nicely add up to 1.

So we then exponentiate them and then divide by the total, which is basically softmax. And when you do that, you have nice fractions that add up to 1. And then we said, well, the contextual embedding for W6 is just all these weights, S1, S2, all the way to S6, multiplied by the original Ws, and then you get the contextual embedding for W6. So this is the basic logic we covered last time.

Now it is obviously the case that we explained it only for one word, but we have to do the same exact operation for every one of the other words, too, so that we could calculate W5 hat, W4 hat, W3 hat, and so on and so forth. So there's a lot of computations that are going on, and they all look kind of similar, where you've got to do a bunch of dot products, you've got to do some softmaxing on it and stuff like that. So the natural question is, is there a way to organize it very efficiently?

And the short answer is yes. In fact, if you could not do that, there wouldn't be any transformer revolution because it is that ability to package it up into a very interesting and efficient operation that allows you to put the whole thing on GPUs. So now I'm going to switch to iPad and give you some iPad scribblings of mine, which were concocted last night because I was very unhappy with the slides that follow. So if you're going to do iPad.

So if it works, you folks are lucky. If it doesn't work, last year's HODL class is luckier. So let's shift to that. So we're going to go here. So let's assume we have a simple thing like-- instead of, "Train left the station," which is a long sentence, let's just say you have a simple sentence like, "I love HODL." So "I love HODL" is what you have, and then you have these standalone embeddings, W1, W2, W3.

So it comes into the self-attention layer. And let's assume that these W1s, Ws-2, W3, they are already positionally encoded. We have already added up the positional encoding all that stuff. It's all behind us. That all happens outside the transformer. So you get it here. Now what you do is you actually make three copies of this thing. And let's call this whole thing as just x. I'm just giving it the name x. It's a matrix of these three vectors.

And so the first copy goes up here. The second copy goes straight. The third copy goes down. And don't worry about the third copy just yet. So if you look at the first two copies, here is the key thing to focus on, this whole thing here. Remember that we want to calculate dot products between all these vectors, and basically we want to calculate the dot product of every pair of vectors, every pair of words.

The whole point of self-attention is that every pair of words, we figure out how attracted or related they are, which means that we have to calculate all pairs of dot products. And so what you do is you take this vector right there, W1, W, W3, you take this other copy that went up, and then you transpose it. So when you transpose it, it all becomes nice and vertical like that. All the vectors came in like this. When you transfer it, it becomes vertical.

And now what you do is you take each one. You take W1 and then you multiply it by W1 here. You take W1, W2, W1, W3. You calculate all those dot products like that. And when you do that, you have these nice cells where every pair of words, the dot products have been calculated in this grid. And the key thing to see here-- and folks with a matrix algebra background will see this immediately-- all we are doing is we are taking this x, which is the matrix that came in, and then X transpose, which is the matrix that we sent up and then brought back down. We are basically doing a matrix multiplication of x times x transpose. That's all we are doing.

And when we do that, we're getting this nice grid in which every pair of words, their dot products have been calculated for you with one matrix multiplication. Boom, done. So if you have three words, there are 9 multiplications. So if you have a million words, that's a lot of multiplications, one trillion multiplications, on the order of one trillion. And the reason I say order of is because W1 times W3 is the same as W3 times W1, so there is some duplication here.

So you get this grid in one shot, in one multiplication. And then because each of these numbers is just a dot product, which can be negative or positive, we need to softmax it. And so what we do is we take all these numbers, and we put it into a softmax function where for each row, it calculates the softmax. And what do I mean by that? It takes each number here, does e raised to the top, e raised to the number, it does it for each of these numbers, and then divides by the sum of those numbers for each row. And when you do that-- you can think of this operation as softmax applied to x times x transpose-- you get this nice little table of numbers.

This table of numbers basically says that for the first word, W1, for the first word, take 0.1 of the first one, 0.7 of the second, 0.2 of the third, and add them up. We do a weighted average. So we have this table here. We have now the third copy, shows up here. It's right there. So we do this times that, which is just a matrix multiplication again. And when we do that, we get the final contextual embeddings.

So this, for example, is just 0.1 times W1, 0.2 times W2-- sorry, 0.7 times W2 and then 0.2 times W3 right there. And you can see the same logic here as well. And you can read it later on. I will post this thing to make sure you understand exactly how it flowed. But the larger point I want you to focus on is that the entire self-attention operation we just looked at here basically is this beautifully little compact matrix formula. x comes in, you do x transpose, you do a matrix multiplication, you do a softmax on top of it, and then multiply it by x again and boom, you're done.

So that is the magic of taking the transformer stack and representing it using matrix operations because then [POOF], lightning fast on GPUs. OK? All right, that was the warm-up. Now let's crank it up a notch. So recall that in the last class, I talked about the fact the self-attention operation, the Ws are coming in, and we're doing all this stuff with the Ws, and then we're getting some W hats out, but there are no parameters.

There IS nothing to be learned inside the transformer self-attention layer, right? There are no weights. There are no biases. There are no coefficients. Well, OK, what are we learning then? So what we now do is we are going to make the self-attention layer tunable. We are going to inject some weights into it so that when we train it on an actual system, the weights will keep changing to adapt itself to the particularities of whatever problem you're working on.

So that takes us to the tunable self-attention layer, tunable self-attention layer. So this is the key thing to keep in mind. Any questions on this before I continue with the tunability thing? OK. Is this picture working out, by the way? OK. So what we now do is we have the same exact logic as before, where we have this thing that comes in. We have this input that comes in. We call it x again, this matrix of embeddings. And then before we just send three copies, instead of doing that, what we're going to do is we'll take each copy, x, and then we will actually multiply it by a matrix.

This matrix is called the key matrix. And this matrix, this matrix of numbers or weights that will be learned by backprop. So basically what we are saying is that when this thing comes in, let's see if there is a way to transform this x into some other set of embeddings which may be useful for your task. We don't know if they're going to be useful, but surely giving it a bit more ability to have weights which can be learned means that we are giving it more expressive power, more modeling capacity.

And whether it actually uses the capacity will depend on how much data you have and how well you train it. And maybe if it's not useful, it won't use it. What I mean is if transforming x actually doesn't really help at all, then this matrix A is going to be what?

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: It's going to be the identity matrix because you take basically 1 and multiply it by x, you'll get x again. So the worst case, maybe it just says, I have nothing to learn here, but maybe there is something you can learn. So that's what we do. So we multiplied by this matrix AK. And then we come up with the same-- some embeddings, transformed embeddings, and we call these things K.

Now this KQV, as you will see, has its origins in the field of information retrieval, but I personally find that interpretation is not super helpful because transformers are used for lots of applications outside information retrieval, so I'm not going to go with that kind of interpretation. I'm going to go with the interpretation of, let's make each of these things tunable. And tunability means we need to give it weights.

So that's what we have here. Now the second copy-- we did this with the first copy. Well, let's do the same thing with the second copy. We'll take the second copy and multiply it by some other matrix called AQ. And when we are done with that, we get these embeddings, and we will call these embeddings as Q.

Now just like before, we will take this thing here, and we will transpose it so it all becomes nice and vertical like that. And then we'll do exactly the same as before. We'll calculate all these pairwise dot products using one-shot one-matrix multiplication. And because we are calling this Q and we are calling this whole thing as K, this thing just becomes Q times KT.

At the end of it, you come up with a grid of numbers just like before. And these numbers could be negative or positive, so we need to do the softmax on them to make sure they are well-behaved fractions that add up to 1. So we take this QKT business and then we put it through a softmax function for each row. And when we do that, we will get basically a table like the ones we saw before.

By the way, the numbers here are the same just because I duplicated it because I'm lazy. In reality, given that it has gone through all these transformations, the numbers are not going to be the same. You have these numbers and then you take the final copy, which is X times AV. Each copy is getting multiplied by its own matrix. And this copy is being multiplied by AV.

And let's call this XAV, which is here, as just V. And so what you have here is this softmax QKT times V is exactly the same kind of dot product as we saw before, matrix multiplication. So we have these contextual embeddings, and that's what's coming out of the transformer block. So now the whole thing we did here, the whole thing can be represented as softmax of QKT times V.

So if we zoom in a bit-- come on. So X came in, three tracks went here. The first track X times AK, X times AQ, X times AV. And this thing is called K. This thing is called Q. This thing is called V. And then we do the same transpose as before. We do the dot product thing to calculate the pairwise dot products for everything, which is just QKT.

We run it through a softmax. We get softmax of QKT. We multiply it by 1 to do the final weighting, and then boom, the output comes, and that's this function. That's it. So what we have done is we have introduced three matrices, learnable matrices, into the self-attention layer. Let me just stop there for a second. Questions? Yeah.

AUDIENCE: Is there a relationship between AK, AQ, and AV?

RAMA RAMAKRISHNAN: Independent.

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: Independent matrices. Yes?

AUDIENCE: [INAUDIBLE]

RAMA RAMAKRISHNAN: Could you use the microphone, please?

AUDIENCE: Here we have three sets of parameters, K, Q, and V. If there are, let's say, if there were 100 or the total length was, let's say, the number of totals, let's say 50, so you could have 50% of parameters you have to [INAUDIBLE].

RAMA RAMAKRISHNAN: So if the dimension is 50 long, what is coming in, the Ws are 50 long, then the key, what comes out of it, if you want it to be 50 as well-- so this matrix needs to be 50 times 50, 2,500.

AUDIENCE: What are the three different things the three matrices are trying to learn?

RAMA RAMAKRISHNAN: Sorry?

AUDIENCE: What are the different things that the matrices are trying to learn?

RAMA RAMAKRISHNAN: We don't know. All we are saying is that we have a self-attention layer, which can pay attention to every pair of words, but we need to give it some ways to transform what is coming in into potentially useful things. As to their actual usefulness, we'll have to figure out if it actually helps or not. And of course, as you know, the punchline is that yeah, it helps massively. That's why we do it.

In general, what you will find in the deep learning literature is that whenever you want to increase the capacity, the modeling capacity of a particular model, you just take a small piece and inject a little matrix multiplication into it. You take a vector that's showing up in the middle, and then you make it run through a matrix to get another vector. And then further, after you run it through a matrix, you run it through a little ReLU as well. Even better. So that's how you inject modeling capacity into the middle of these networks, and that's what these people are doing here. Yeah?

AUDIENCE: In the last step, you had the matrix V. So on the previous example, you had used the original matrix X. Could you just say [INAUDIBLE] why is it not using X? And what does that mean?

RAMA RAMAKRISHNAN: So what you are saying is that in the initial version we had three copies, and we treated them all identical. Now we said, well, are there ways to transform each copy into some other representation which could be useful, so we may as well use three different matrices for it. Why stop with two? There are three opportunities to make them more expressive. We'll use all of them. Yeah?

AUDIENCE: You mentioned that these are-- you're fine-tuning it. Is there any risk--

RAMA RAMAKRISHNAN: We're not fine-tuning it, just to be clear on the vocabulary here. So we have added more weights to make them tunable. What that means is that when we finally train this entire model, remember, all the weights are going to be updated using back propagation. In particular, these matrices will also get updated using back propagation.

AUDIENCE: So there's no risk of over-- is there a risk of--

RAMA RAMAKRISHNAN: There is always a risk of overfitting when you add more parameters to a model, which means that you have to look at the validation set and all that good stuff. We are basically adding more parameters in a very interesting way because we want to add more capacity to the self-attention layer. We want to give it more of an ability to learn things from the data. Before it could not learn anything. It could only do dot products. So we want to solve that problem. All right, I'm going to continue, and we'll come back to this.

Just for fun, I'm going to do this. The original paper is called, "Attention is All You Need." This is a transformer paper. You folks should read it at some point. I just want to show you something. You see that? So that is the famous transformer formula. And the only thing we ignored is this root of dk business under it.

I wouldn't worry about it. The reason they have it is because these softmaxes, when you have lots of numbers and some numbers are really, really big, what's going to happen is that all the other numbers are going to get squashed to 0. And so to make sure the gradient flows properly, they just divide it by a particular number to make sure no number is too big. It's a small technical-- important, but a bit of a technical detail, which is why I ignored it in my iPad.

But the rest of it, you can see this is exactly the formula we derived, QKT times V softmax. So this is the famous transformer formula. And congratulations, now you understand it. You seem less than fully convinced? [CHUCKLES] OK. Yes, iPad. Now I have a bunch of slides, which I had-- actually, I'll come back to this.

I had a bunch of other slides-- this is from last year-- which actually explains what I did in the iPad in a very different way without using any matrices and so on. I was looking at it last evening, and I was getting very annoyed by these slides for some reason because I felt that it wasn't really conveying the ability of using matrix algebra to actually do this so efficiently and compactly, which is why I decided to hand-draw this thing on the iPad. But you should read it afterwards to make sure that whatever you saw on the iPad actually matches this because two different ways of understanding something always helps.

So this is what we have here. Now to just to recall, by making self-attention tunable, we get a very interesting benefit, which is that when you have these different attention heads, before, you could have two attention heads, but because there were no parameters inside, their outputs would have been identical. Because the inputs are the same for both. Therefore, the outputs will be identical. But now since each attention head will have its own AQ AK AV matrix, the outputs are going to be different. That's why it makes sense to do the tunability thing because that's what actually makes multiple attention-- it's actually useful.

AUDIENCE: Is there actually any relationship between AK, AQ, and AV, or is the A just for a notation standpoint?

RAMA RAMAKRISHNAN: Just notation. The thing is we want to use QKV for the resulting matrix, and so I had to find something else to use for the first one. And I was like, OK, AQ AQ. And we are at MIT, we do subscripts, superscripts. Yeah?

AUDIENCE: What is the size of the matrices? They're like square matrices or?

RAMA RAMAKRISHNAN: Yeah. So typically what happens is that there is a whole bunch-- you can think of it as a hyperparameter in some ways. Typically what people do in most implementations is that they will actually just preserve the size. So if the incoming embedding is 10, they'll make sure the thing coming out of the thing is also 10. So you just do a 10 by 10 matrix to transform it.

But the value V AV matrix, on the other hand, there is a bit more technical stuff going on where it often tends to be smaller. So for example, let's say that your incoming is 100. You do 100 to 100 for the key, 100 to 100 for the query. But if you have say, 5 attention heads, you may do 100 to 20 for the Vs because ultimately all the Vs are going to get concatenated into another 100 again. So I can tell you more offline. But broadly speaking, these things tend to get transformed. They preserve the dimension, 10 in, 10 out. Yeah?

AUDIENCE: So this AQ AV, these numbers are random you start with it and then [INAUDIBLE].

RAMA RAMAKRISHNAN: Exactly. Exactly. So the values in these matrices are weights learned through optimization using SGD. And then what that means is that each of these attention rates now has its own copy of these matrices. It has its own matrices. And over the course of backpropagation, these matrices will look very different. So important, each attention head will have its own set of three matrices. So if you have 10 attention heads, 30 matrices will be learned.

AUDIENCE: So by the math, it seems like it's creating essentially a relationship between all of the content being ingested. And if you're ingesting all the content for each attention head, are there different categories of attention head type that you're trying to go after?

RAMA RAMAKRISHNAN: Yeah. So basically what we are trying to do is to say a particular attention head-- so in any particular sentence, it may turn out to be the case that one pattern could be about the meanings of these words, like the word "bank" and what it means, the word "station," "train," things like that. That's what really we've been talking about. But there is a whole other pattern to do with grammar and tense and things like that. There could be another one in terms of tone.

All those things are very important, and a priori we don't know how many such patterns exist. Much like in a convolutional network, when we are designing how many filters to have, we don't know how many kinds of little things we have to detect, vertical line, horizontal line, semicircle, quarter circle, stuff like that. So you just give it a lot of capacity so that it can learn whatever it wants. So that is the transformer encoder. So we have done the first of the three complications needed to make it industrial strength and legit.

The second thing we do is something called the residual connection. So what we do is that whatever comes out here, W1 through W6, goes in and comes out as W1 hat, W2, and so on and so forth. Actually, sorry. What comes out here is the hats. But what comes out here is some intermediate Ws. That is what the self-attention is going to give you, some intermediate Ws. What we do is-- and because what's coming out here, these vectors are the same length as what goes in, we can just add them element by element. So we take the input and we actually add it to what comes out.

So why would we want to do that? Why would we want to go to a lot of trouble to process this thing, and then when it comes out, we literally add up the original input? What do you think the intuition is? So it turns out-- think of it this way. You have a bunch of inputs. You send it to a neural network. It transforms it and gives you something else.

At that point, you might be thinking, well, everything that happens in the network from that point onward can no longer see your original input. It can only work with the transformed input, right? But what if your transformations are not great? So as an insurance policy, what you can do is you can take the transformed stuff and you can take the original stuff and send both in.

And this whole thing-- and you can Google it. It's called a wide and deep network and things like that. But the whole point is that, let's not lose the original input anywhere. Let's also send it along. But if you keep adding the original input to every intermediate layer, it's going to get longer and longer and longer and bigger, which you don't want because you want it all to be the same size. So the simplest alternative is to just add them up.

You take the transformed stuff and you add the original input, you get the same thing again. What came in, W1 was a 100-long vector, and the transformed version is also 100 long. So just literally under 100, add them up. That's it. You get another 100-long vector. So that is what's called a residual connection. And as it turns out, residual connections improve the gradient flow during backpropagation dramatically, and that's why they are very heavily used.

And in fact, ResNet, which we looked at for computer vision, it stands for Residual Net because it was the first network to actually figure this out. This is not just a transformer thing, by the way. It's widely used in lots of deep neural architectures. The notion of a residual connection, that's what it means.

So we do a residual connection, and then we come to the final tweak, which is called layer normalization. So once we add the residual connection, we are going to do something else here to these vectors before they continue flowing. And what layer normalization does is it basically says that-- you will recall from the very beginning of the semester, I've been saying that whatever comes into a neural network, the inputs, let's just really make sure that they are all in some sort of a narrow, well-defined range. They can't be in a big range.

So for pictures, for images, we divided every number by 255 so that every little pixel value is between 0 and 1. For continuous things, like the heart disease example, we standardized it by calculating the mean and the standard deviation and subtracting the mean and dividing by the standard deviation. So when you do that, all the numbers are going to roughly be in the minus 1 to plus 1 range.

So in neural networks, for a backprop to work really well, you have to make sure that no numbers get too big, that all the numbers are always in some sort of a narrow range. So what layer normalization does is to say, you know what, whatever is coming out here, I want to make sure none of these numbers are too big. I want to make sure they're all well-behaved in a small range because if I don't do that, backprop is not going to work very well.

AUDIENCE: Is this what we do to ensure we don't have the problem of vanishing?

RAMA RAMAKRISHNAN: Right. So technically there could be two problems. There is an exploding gradient and a vanishing gradient. Both are bad. This is a way to address it. So you will find a whole bunch of batch normalization techniques, layer normalization, batch normalization, and so on and so forth. All these are methods to make sure that these numbers stay in a small range so it doesn't cause gradient issues later.

In particular, what we do is-- or what happens inside this layer normalization-- is we just calculate the mean and standard deviation of every one of these embeddings If you have, let's say, 6 embeddings here, we'll have 6 means and 6 standard deviations for each one across the rows. And then we standardize it, meaning subtract the mean, divide by the standard deviation. And when you do that, all these things are going to be nice and small.

And then we do this little other thing where we have introduced two new parameters to rescale it and move it around a little bit just because adding more weights always helps make these things better, so we add them. And this gets slightly complicated because of the way the dimensions work, so I'm not going to spend much time on it. And then what comes out the other end is a very well-behaved set of numbers and a nice and small and narrow range. So this is called layer normalization. You can see this link to understand it a bit better. And we do that as well.

So to put it all together, so this is a transformer encoder where we have this multi-head attention layer where each attention head in the inside of it is tunable with those A matrices, and then we have a residual connection. We do that, and then we do layernorm. And then we do the same thing in the next feedforward layer as well and then boom, out pops the output.

AUDIENCE: By that definition, the multi-head attention layer [INAUDIBLE] and everything, theoretically I can add even the biases or the hate speech aspects which come in to take care of it. So the model can account for the fact that something is biased or something is not.

RAMA RAMAKRISHNAN: The thing is it's not so much the model's accounting for it. It is capturing whatever patterns happen to be inherent in the data. It's capturing. Now what you do with that capture is up to you. It depends on the actual problem you're trying to solve. In particular, it is going to capture all the bad stuff, too. Because if your training data has a lot of biased stuff in it, toxic things in it, dangerous things in it, it doesn't have a sense of values as to what is good or bad, it's just going to pick it up. Yes?

AUDIENCE: Picking on that, then how do you actually make it unlearn those or how do you mitigate the effect of those?

RAMA RAMAKRISHNAN: That's a whole course unto itself, but I'm happy to give you pointers offline. So this is what we have. And remember what I said, that this is just a single transformer block. And since what comes in and what goes out are the same dimensions, we can just stack them one after the other. It's very stackable. You can stack it vertically as much as you want. And as I mentioned, I think GPT3 has 96 of these things stacked one on top of the other.

That is the transformer encoder, and this exactly maps to that. So basically the input embeddings come in, you add positional embeddings, and then you send it to, say, these mini attention blocks. And they all get added up, and then it comes out to the attention block. You add-- the add and norm here means-- "add" means residual connection because you're adding the input, which is why you have this arrow going from the input being added there, and then you normalize it, send it along, and do it again, and out comes the output.

Now just to be very clear on what is being optimized during backpropagation in this complex flow, now, clearly the embeddings that you started out with, both the standalone embeddings as well as the position embeddings, those things are going to get optimized. Those are just weights. They're going to get optimized. Clearly everything inside the transformer encoder block is going to get normalized.

And what are they? Well, they are the AK, AQ, AV matrices for each attention head. Layer norm has parameters as well. The little feedforward layer has weights as well. All these things are going to get optimized. And then it goes through this ReLU which, again, has a bunch of weights. That's going to get optimized. And then the final softmax has a bunch of weights. That's going to get optimized. All these things are going to get optimized by a backprop.

So in that sense, you just step back for a second and look at the whole thing. It is just a mathematical model with a lot of parameters. And we're just going to use a gradient descent or stochastic gradient descent optimizer. That's it. Yeah?

AUDIENCE: [INAUDIBLE] we train the model. Are we calculating weights for each cell of every possible matrix based on the number of inputs, every possible dimension up to the max number of inputs?

RAMA RAMAKRISHNAN: Actually, the weights themselves don't depend on how long your input sentence is. Because remember, what we're doing is for each sentence that comes in-- let's say the sentence has, say, three words. There are three embeddings for that sentence. Each of those embeddings gets multiplied by, say, AK.

So AK only needs to know how long is each embedding. It doesn't need to know, how many words do I have. And I'm glad you raised that question, Ben, because that's what makes the transformers' number of weights independent of the number of words in your sentence. It only depends on the vocabulary that you're going to work with because the vocabulary determines how many embeddings you need, how many embeddings you need.

The length only matters in terms of the positional embedding because if you have 1,000-long sentence, you need 1,000-long positional embedding matrix. But beyond that, it doesn't care. And that's why, for example, Google Gemini 1.5 Pro, which can accommodate basically 1 million long, 1 million token context window, it's still very compute-heavy, but it does not change the number of parameters. Yeah?

AUDIENCE: Essentially, which weights are optimized first? Are they in sequential order or are they optimizing the weights at the very same time [INAUDIBLE].

RAMA RAMAKRISHNAN: Simultaneously. Because if you think of backpropagation, ultimately, you have a loss function, and you calculate the gradient of that loss function. So if you have, say, 1 billion parameters, that gradient is basically a 1-billion long vector. And we're going to take the gradient, and we're going to do W new equals W old minus alpha times that gradient. So all the Ws are going to update instantaneously.

Now the way it actually works in computation is you're going to do it-- because of the back and backpropagation, it's going to start at the end and slowly flow backwards, but when it's done, everything will be updated. Yeah?

AUDIENCE: [INAUDIBLE] we take two attention heads and we have the matrices of AK, AQ, and AV. [INAUDIBLE] the parameters of all three of them, all the weights of the three matrices on this side and this side would be different because finally, the things you are inputting from this side and the output is same. So the learning process should be ideally the same unlike like a CNN, where we had put filters which were different. So what different thing we have

RAMA RAMAKRISHNAN: Because the initialization is different.

AUDIENCE: Oh. What do you mean?

RAMA RAMAKRISHNAN: What I mean is if you have two heads, each head has three matrices. The starting values of those six matrices are different.

AUDIENCE: If the starting value of AK, AQ, and AV is different on both heads.

RAMA RAMAKRISHNAN: Correct. Much like for all the weights, typically, the values are randomly chosen. If they were all the same thing, you're right, it won't make a difference. But they will all change the same way. Yeah?

AUDIENCE: Is the input of the transformer the sentence or the array of the embedding of each word?

RAMA RAMAKRISHNAN: The transformer itself is expecting embeddings in. And so what basically happens is that we get some sentence, we run it through a tokenizer which connects it to a bunch of tokens, which are just integers, and then it goes through the embedding layer, which maps the integers to these embeddings, and then you feed it to the transformer. But when you do backpropagation, it comes all the way back to the starting embedding layer and updates those weights.

AUDIENCE: So it can be trainable so the embeddings that we set at the beginning, we see them as input here, but they can be trained.

RAMA RAMAKRISHNAN: They are trainable, exactly. Exactly. Yeah?

AUDIENCE: Are the attention heads solely parallel or can you have a stack of attention heads?

RAMA RAMAKRISHNAN: Typically, they are parallelized because you can always stack the block itself to get more and more power. So now to apply the transformer, common use cases are that you have a whole sentence that comes in, and then you just want to classify it, the canonical thing being hey, movie sentiment classification, boom, positive or negative right. Classification.

Another common one is labeling, where every word gets labeled as a multiclass label, and that's basically what we saw with our slot-filling problem. And then there is another thing called sequence generation where you give it a sequence. You want it to continue the sequence, generate more stuff, i.e. large language models and all that good stuff. So this we already know how to do because we actually literally built a Colab with this, with the transformer stack.

Now the question is, how can we do that? How can we do basic classification with these things? So now again, when you send a sentence in, after all that stuff is done, and when I say, encoder, here, I'm assuming that you may have one block. You may have 106 blocks, I don't care. At the end of the day, you send something in. You get a bunch of contextual embeddings out.

So at this point, we need to take these contextual embeddings and somehow make it work for classification. We are just classifying something into yes or no, positive or negative. So it would be nice if we can actually take all these embeddings and essentially summarize them into a single embedding, a single vector. Because if we have a single vector, then we can run it through maybe a ReLU, and then we do a sigmoid, and boom, we can do a binary classification problem. Super easy.

So this begs the question, OK, how are we going to go from all the many blue things to one green thing? Now of course, what we can do is we can simply average them. We can take each of the embeddings and just simply average them element by element. You'll get a nice green thing. Any shortcomings from doing that?

AUDIENCE: You would lose the ordering of the words.

RAMA RAMAKRISHNAN: Well, in some sense, the positional embedding, the positional encoding you have in the input does have this notion of position. So you're not necessarily losing the order necessarily, but you're sort of averaging all this information into something, and averaging is going to lose some richness.

AUDIENCE: I think it's going to be skewed to the one that has the biggest number, so--

RAMA RAMAKRISHNAN: True.

AUDIENCE: [INAUDIBLE] is influencing your--

RAMA RAMAKRISHNAN: Yeah, the biggest ones are going to dominate. But hopefully we won't have too much of that because all the layernorm business in the beginning has hopefully made sure the numbers are all in a reasonably small and well-behaved range. But the point really is that you're going to lose richness in the information because you're just pssht, mushing it down.

So there's a much better and more elegant way to do this, which is that what you do is for every sentence, when you train it, you add an artificial token called the class token. Literally, it's an artificial token and it's designated as CLS in the literature. And then this token is getting trained with everything else. And so once you finish training, that token has its own embedding, too. And because it has been trained with everything else--and this token, remember, is a contextual embedding, which means that it's very much aware of all the other words in the sentence.

So in some sense, this CLS token's contextual embedding captures everything that's going on about that sentence. And so what we do is once we are done training, we just grab this thing alone and then send that through a ReLU and a sigmoid and boom, you're done. So this is a very clever trick to somehow, instead of averaging everything at the end, let's just have something just for the whole thing, the sentence, and just learn it anyway along with everything else.

A meta principle in deep learning is that whenever you think you're making an ad hoc decision about something, like averaging a bunch of stuff, you should always stop and say, is there a better way to do it where it doesn't have to be ad hoc, where the right way is learnable from the data directly using backpropagation? There was a hand. Yeah?

AUDIENCE: Is there a reason that you added the CLS at the start? Why not earlier? [INAUDIBLE]

RAMA RAMAKRISHNAN: You can do it that way.

AUDIENCE: At the end, is there any difference?

RAMA RAMAKRISHNAN: The only thing to remember is that-- it's a good question. So different sentences are going to be of different length. So there might be short sentences. There might be long sentences. In particular, the short sentences are going to get padded. I remember I talked about padding to make it to fit to one length.

So what internally the transformer will do is it will ignore all the padded tokens because it doesn't do it. It's just padding. It doesn't really matter for anything. So if you have the CLS at the very end, we have to have much more administrative bookkeeping to take everything but the last one, ignore it, and only do the last one. It's just much easier to stick it in the beginning. That's the reason. Yeah?

AUDIENCE: What could be just a practical application of this? Would it be something like sentiment analysis--

RAMA RAMAKRISHNAN: Exactly.

AUDIENCE: --like positive or negative?

RAMA RAMAKRISHNAN: Yeah. So basically any kind of text comes in and you want to figure out some labeling problem, like a classification problem. The easiest example I could think of was sentiment. But you can imagine, for example, an email comes into a call center operation, and you want to take the email and automatically figure out, which department should I send it to.

Now if the input data for a task is natural language text, we don't have to restrict ourselves to only the input training data we have. Wouldn't it be great to learn from all the text that's out there? So for example, to go back to that call center thing I just mentioned, let's say it's coming in English. The ability to take that English email and route it to one of 10 things, you shouldn't have to learn English just for your call center application. You should learn English generally and use it for other things. So why can't we just learn from all the text that's out there?

And so that brings us to something called self-supervised learning. And the idea of self-supervised learning is this. So if you recall the transfer learning example from lecture four where we had ResNet, and we took ResNet, we chopped off the final thing, we made it headless, and then we attached that output of the headless ResNet to a little hidden layer and output, and we did the handbags and shoes. And you will recall that we were able to build a very good classifier for handbags and shoes with just 100 examples.

So the question is, why was this so effective? Why was it so effective? And turns out the reason why any of this stuff actually works is because neural networks, they learn representations automatically when you train them. So what I mean by that is when you imagine a network, you feed in a bunch of stuff, it goes through all the layers, it comes out, you can think of each layer as transforming the raw input in some different alternate representation of the input. And these are called representations. That's actually a technical term.

And so from this perspective, when you train a neural network, a deep network with lots of layers, what you're really learning is you're learning how to represent the input in many different ways-- each of these arrows is a different way of representing things-- plus you're learning a final regression model, either a linear regression model or a logistic regression model. Fundamentally, that's what's going on.

Because the final layers tend to be sigmoid, softmax or just linear, right. So the final layer-- if you just look at this part alone, whatever is coming in is just going through essentially a linear regression model or a logistic regression model. That's it. So fundamentally you're learning representations and a final little model. But the reason why all these things work so much better than logistic regression is because those representations have learned all kinds of useful things about the input data. They have sort of' automatically feature-engineered for you.

So from this perspective, you can imagine that each layer here is like an encoder. It encodes the input. The first layer encodes it. The first two layers encode something. The first three layers encode something, and so on and so forth, so a deep network contains many encoders. And so the question is, what do these representations actually embody? What do they capture? Is it specific knowledge about the particular problem that you train the network on? Or is it general knowledge about the input data?

Because if it is general knowledge about the input, we can use it to solve other problems, unrelated problems, so is it specific knowledge or general knowledge. And it turns out they actually capture a lot of general knowledge about the input. And that's why you can get reuse out of them. You can reuse them for other unrelated things because they have captured general stuff.

So if you look at this-- I think I've shown you before. If you look at a network that classifies everyday objects into a bunch of categories, it can learn all these little patterns in the beginning and later on and so on and so forth. And this is a face-detection network. It has learned how to identify little circles and edges and nose-like shapes and finally faces. So all these things are examples of representation learning interesting things about the input.

So since these representations are capturing intrinsic aspects of the data, you can use it for other things. You can take a face detection neural network and reuse it for emotion detection, for instance. So the question is if we can somehow get an encoder that generates good representations for your input data, we can simply build a regression model with those as input and labels as output and be done.

And this is exactly what we did with ResNet for handbags and shoes. We found a thing that had already been trained on similar everyday objects, everyday images. And the key insight here is that since we don't have to spend precious data on learning these good representations, we won't need as much labeled data in the first place because the pre-training used a lot of data, and you're piggybacking on that data. So in some sense, your training data is everything that the pre-trained model was trained on plus your little 200 examples.

So this is what we did. We used headless ResNet as an encoder that can take raw input and transform it into useful representations. This is what we did. So the general approach is that you find a deep neural network built on similar inputs but different outputs, and then you basically grab maybe the penultimate representation or the one before that. Then you chop off the head. You attach your own output head, train the whole thing-- just the final layer or train the whole thing if you want. This is like the playbook we followed for ResNet.

The same thing works for all kinds of other data types as well. So now to build such a model, we need labeled data. We were lucky because ResNet was actually trained on ImageNet data, which is like 1 million images, each of which labeled into 1,000 categories, which is very convenient for us.

But what if you want to build a generally useful model for text data? Clearly, we need to collect a lot of text data, but that's no problem because the internet is full of text data. We can easily scrape the internet. We can just download Wikipedia. So that's not a problem. The problem is something else, which is that, how do we define an input label for a piece of text? So for an input sentence, what should the output label be? That's the key question. Because if you can answer this question, you can just pre-train all these things on all kinds of text data.

So a beautiful idea for doing this is called self-supervised learning. And the key idea is that you take your input, whatever the input is, you take a small part of the input and just remove it, and then ask your network to fill in the blanks from everything else. So this is called masking, and it's just one of many techniques in self-supervised learning, but this is very commonly used.

So this is the original input. And then you take it and then you just take this thing in the middle here randomly and zero it out or mask it. And so this incomplete input is your now new input, and the thing that you took out becomes your fake label. So you can almost imagine if you're baking donuts, you make a donut, and then you punch a hole in the middle of the donut. The donut with the hole is your new input. The munchkin is the label. Am I making everybody hungry at this point?

[LAUGHTER]

So once you do that, no problem. You have an input. You have labels. You just train a neural network to essentially predict those-- to basically fill in the blanks. And so for example, if you take a sentence like, "The Sloan School's mission," you can just go in there and just knock out randomly a bunch of words like-- the ones I'm knocking out, I'm just putting the word "mask" in it just to show what I'm doing. And then given this sentence, it will try to fill in the blanks with actual words.

So now for the amazing part. In the process of learning to fill in the blanks, the network learns a really good representation of the kind of input data it's seeing. And it makes sense because if I give you a sentence with a few missing blanks and you're able to very successfully fill in the blanks, you have learned a whole bunch of stuff about the world to be able to do that. If I say the capital of France is dash, and you're like, Paris, OK, how did you know that? It's sort of like that. By learning to fill in the blanks, you really have to learn how all these things work, all the connections between various words and so on and so forth.

So what you can do is once we build such a model, we can just extract an encoder from it, and then we'll fine-tune like we did for transfer learning. But this is how you build a generic, a generic pre-trained model on unlabeled data. And so we can use a transformer encoder to build this whole thing in the middle. Because remember, the transformer can take any sentence and give you the same-size sentence back along with predictions for everything, so we can just have it take this thing in and ask it to just predict all the missing words here.

So to put it in other words, masked self-supervised learning is just a sequence labeling problem. So basically this is the sequence that comes in, and then you tell the transformer-- and you get all these embeddings. It goes through all that stuff. You really don't care about these outputs. But wherever the word "mask" went in in the input, you basically try to get it to-- the right answer is, for example, the word "mission," and that is the right answer. This is the right answer here. And then you take these right answers, create a loss function, and do a backprop, and boom, you're done. Inputs right answers and you're in business. That's it.

Now if we pre-train a transformer model like this on massive amounts of English text-- let's say we did that-- we get something called BERT. BERT is a very famous transformer model. And BERT was the first model, actually, that Google used to upgrade its search in 2019. The Brazil visa example, you may recall from earlier lectures, that uses BERT under the hood.

Now I just want to show you-- because you can actually read the BERT paper and it'll actually make sense to you now based on what you've learned in this class. Look at this. BERT's model architecture is a multi-layer bidirectional transformer encoder. OK, transformer encoder. We denote the number of layers, transformer blocks, as L, the hidden size as H, and the number of attention heads as A. And how much is that?

We want H as 768, which means that the embedding sizes are 768 and the hidden feedforward layer is four times as much, so it's 4,096-- sorry, 4,096 is the feedforward layer. The embeddings are 768. And you can see there are two BERT models here. This one has 12 transformer blocks. This one has 24 transformer blocks. So you can actually read this paper. You can actually relate it to exactly what we discussed in class. It'll all make sense.

Bidirectional essentially means that the words can pay attention to every other word in the sentence. And as we will see on Monday, you have another transformer thing called a causal transformer, in which you only pay attention to the words that came before you, not the ones after you. So bidirectional means all words are seen.

So what we do is, remember, we said to solve sequence classification, you can add a little token at the beginning and then boom, use it for classification. As it turns out, very conveniently for us, the people who built BERT, when they trained BERT, they just used the CLS business during training, so it's actually available for us out of the box. So when you use BERT for sequence classification, you don't even have to do any surgery on it. It just gives you the class token automatically, which is very convenient.

And you can also use it for sequence labeling as well. So for sequence classification and sequence labeling, BERT is actually usually a really good starting point. And in particular, there have been lots of improvements and variations of BERT over the years. And if you're curious about this, there's a thing called The Sentence Transformers library, which has got a whole bunch of BERT-related code and resources that you can use to do things out of the box.

OK, this is a bit of a word wall. So to solve any of these problems, classification or labeling, where the input is natural language, we can obviously use a model like BERT, label a few hundred examples, attach the right final layers, and fine-tune it, like we did for the ResNet. But if your problem is like a standard NLP problem, you don't even have to do that because people, for these standard tasks, they've already pre-trained it on those standard tasks. And so you can do all these things without any fine-tuning at all, literally out of the box.

And so there are many hubs which have these pre-trained models. But perhaps the biggest one is the Hugging Face Hub. And I checked last night. It has 520,000 models available. I think if I recall, last year when I taught HODL, I think the number was a lot smaller, maybe 50,000, so it's growing really, really fast.

Let's just switch to Hugging Face Colab. So Hugging Face. How many of you are familiar with Hugging Face? OK, that's good. So for the others, basically, you have a whole bunch of pre-trained models on Hugging Face. You actually have a lot of data sets you can work with for your own tasks. There are lots of people demoing what they have built in this thing called Spaces. And of course, there is a lot of documentation and so on.

So the thing you can do is-- what they have done is they have organized all these models by the kind of task you can use them for. So you can see here, there are a whole bunch of computer vision tasks that you can use them for. There's a whole bunch of natural language tasks like text classification, feature extraction, this and that, lots of interesting examples here.

And so what you do is you just literally can go in there and say, OK, I want to do a text classification. You hit it, and then it tells you all the models that are available. Turns out there are 50,000 models just for text classification. And you can look at, OK, which is most downloaded or which is the most liked, and then you can just use them as a starting point for whatever you want to do.

So that is Hugging Face. And so the way you do Hugging Face is-- I'm just connecting it. If you have a problem which the input is a natural language text, the first question you have to ask is, is it standard or not. Is it a standard task or not. If it's a standard task, do not reinvent the wheel. This thing will usually work pretty well. So here we will use this thing called the Transformers Library from Hugging Face, in particular, the pipeline function, to demonstrate quickly how to do this thing.

Fortunately, this library as of this year is pre-installed in Colab, so we don't have to install it. We can just start using it right away. So we'll take this example where you have a bunch of text which says, "Dear Amazon, Last week I ordered an Optimus Prime action figure from your store in Germany. Unfortunately, when I opened the package, I discovered to my horror that I had been sent an action figure of Megatron instead." Can you imagine that person's sheer distress at this?

"So as a lifelong enemy of the Decepticons, I hope you can understand my dilemma. So to resolve the issue, I demand an exchange. Enclosed are copies. Expect to hear from you soon. Sincerely, Bumblebee." They should come up with a better name for this example. Cool, so that's the text we have. So we import the-- this pipeline function is the one that basically gives you the ability to out-of-the-box start using it without any pre-training, nothing like that.

So we download this thing. Oh, wow. I got an A100 today. That happens very rarely. Sorry. So here let's say you want to classify that text You just want to classify it for sentiment. You literally go in there and say pipeline text classification-- that's the task you want the pipeline to do for you-- and you create a classifier It's going to download a bunch of stuff and so on and so forth.

The first time it just takes time to download. And then you literally take the text you have here, and then run it through the classifier as if it's just a little function. You get some outputs. Actually, let's just do this way. Negative. Sentiment is negative with a 90% probability. Pretty good, right? Sentiment classification solved.

So we'll try a few different examples. "I hated the movie. If I said I loved the movie, I would be lying." OK, that's a little tricky. "The movie left me speechless. Incredible." And then I had to add this last thing here last night. "Almost but not quite entirely unlike anything good I've seen." That's not original, by the way. People who have read Douglas Adams will know this famous sentence about somebody drinking some beverage and saying it's almost but not quite entirely unlike tea. So I was inspired by that. So anyway, we'll see what happens.

All right, put it in there. So negative. "I hated the movie." OK, fine. "If I said I loved the movie, I'd be lying." Negative. "The movie left me speechless." It says it's negative, but it could go either way, right? A good classifier would probably have given you a probability around the 50% mark because it's right on the fence. "Incredible." It's positive. And then it got fooled by my crazy long sentence, and it says it's positive.

Now that's classification. Here is one other quick example. So you can actually give it a piece of text. For example, you can take a Reuters news story. You can feed it and say, extract all the company names from it. Extract company names, people names, and things like that. It's called named entity extraction. And back in the day, people would hand-build painstakingly all these very complex systems to do named entity extraction. Now it's just a pipeline away.

So you can take this thing and you can say, create a pipeline for ner, named extraction. For any particular task that you're using, there might be a few additional parameters you can set as a part of the configuration. So we download this pipeline. Perfect. And then we run the output. So it says, OK, good. Amazon is an organization and Germany is a location, LOC, which is nice. So these things have a standard vocabulary as to ORG, LOC, things like that, which you can read up in the documentation.

And then Bumblebee is a person. And then boy, all the Optimus Prime Transformer stuff is all-- it got fooled. It thinks Optimus Prime is miscellaneous, Decept is miscellaneous, and so on and so forth. But you get the idea. You can take standard things like Reuters news stories and so on. You can just feed it, boop. You can get a very good entity extraction right out of the bat. And once you get these entities extracted, then you can put them into a nice structured data table like a database, and then you can run traditional machine learning on it.

And then I had, I think, a few more examples of question answering. Actually, let's just try that. You can actually give it a thing and ask a question about it, and it can actually give you the answer, which gets into the causal transformer thing that we're going to see on Monday, which builds up into large language models because you obviously can give something, you can give a passage to ChatGPT and ask a question and ask it to give you an answer. So it's really in that thing. But just for fun, let's just do that to see if it's any good.

What does the customer want? And the output is an exchange of Megatron. And it's telling you where it starts in the text and where it ends the relevant passage. It's pretty good, right? Because remember, if you have stuff like this, then when you ask a large language model a question, it gives you an answer. You can actually ask it to give you exactly where in the input it found the answer. And because you know these things are going to hallucinate, you can actually look at the input that it's claiming to use and look at what it says and see if they actually match. It's a way to essentially do QA on LLM output.

So that's what we have here. And I have another bunch of questions which I'll ignore for the moment because I want to go back to the PowerPoint. So if you have a standard task, you can just use pipelines and Hugging Face to actually solve many of them out of the box without any heavy lifting. So I mentioned earlier on that transformers have proven to be effective for a whole bunch of domains outside of natural language processing, like speech recognition, computer vision, and so on and so forth. And so I want to give you a couple of quick examples of how to think about using transformers for non-text applications.

The key insight here is that the architecture of the transformer block that we have looked at, amazingly enough, can be used as-is with no changes, no surgery needed, no clever thinking required for any particular application. What is needed, where the clever thinking may be required, is you need to take the inputs that you're working with, and you need to figure out a way to tokenize and encode them into embeddings, which can then be sent into the transformer. So all the action is in taking that input, that non-text input, and figuring out a way to cast them in the language of embeddings. That's the game.

So here is something called the Vision Transformer, which is very famous, actually. I think it may be perhaps the first transformer architecture that was applied to vision problems. So let's say you have a picture. So let's say you have this picture. It is just a picture. So you have to find a way to create embeddings from this picture or to tokenize this picture in some way.

With sentences, you know, "I love HODL," well, obviously, "I," "love," and "HODL" are three tokens. That's pretty trivial to figure out how to tokenize them. But with a picture, what do you do? It's weird to think of tokenizing a picture. So what these people did is that they say, you know what? I'm going to take this picture and chop it up into small squares.

So in this example, they have taken this big picture and chopped it up into nine little pictures. Then you can take each of those nine pictures, each of those nine pictures. If you look at how it's represented, it's just three tables of numbers, the RGB values. So you can take all those numbers, and you just create a giant long vector from it. You have a huge long vector.

And then you run it through a dense layer to come up with a smaller vector. And that smaller vector is your embedding. That's it. But the way you transform the long vector into a small vector is just a dense layer whose weights can be learned.

So what these people did is they said, well, I'm going to first chop it up into these patches, and then I take each patch and do a linear projection. A flattened patch is nothing more than three tables of numbers flattened into a long vector. That's what the word "flatten" here means. And once you flatten it, I'm just going to run it through a dense layer.

So by the way, you will see the words "linear projection." It's a synonym for, run it through a dense layer. So you run it through a dense layer. You get these nice vectors, these vectors. And now you say, well, you know what, I have to take the order of these things into account because clearly, this little patch is in the top left while this patch is somewhere in the middle. The order matters in the picture, otherwise, every jumbled version is going to be the same thing.

So you use positional embeddings. You basically say there are nine positions in any picture. 0, 1, 2, 3, 4, 5, 6, 7, 8. There are nine positions, so I'm going to create nine position embeddings. And then I'm just going to add them up. Then I'm just going to add them up to this embedding, just like we did with words. With words, each word had an embedding. Each position had an embedding. We added them up. Here each image has an embedding. The position of that little patch in the picture has an embedding. We add them up

And then because we want to use it for classification, no problem. We'll have a little CLS token, and then we just run it through the transformer. That's it. And then you get the CLS token, and then you can attach a softmax to it and say, OK, it's a bird. It's a ball. It's a car. That's it. This simple approach actually works, amazingly enough. So that is the vision transformer. And I'm going through it fast just to give you a sense for how these things work. Any questions? Yeah?

AUDIENCE: My question is in case of text we had fixed number of tokens that is the amount of words which could be there in your vocabulary, any English vocabulary. But here if you look at images, they will probably go into trillions and I know we would not upload one image but we take a total set of product images and we try to subset each one of them, each one would have its own weights like [INAUDIBLE].

RAMA RAMAKRISHNAN: There is no notion of vocabulary here. All we are saying is that given any image, we create nine patches, subimages from it. Each of those patches gets passed through a dense layer, and out comes an embedding. So at that point, any image you give me, I'm going to get you nine embeddings out of it. And once I get the nine embeddings, I just throw it into the meat grinder, the transformer meat grinder.

So another example. I think some of you have asked me outside of class, how good are transformers for structured data, tabular data. For tabular data in general, things like XGBoost, gradient boosting works really, really well, so it's good to try them, certainly. I don't think transformers and deep learning networks have any great edge over XGBoost for structured data problems, so it's worth trying both of them.

However, you can use transformers for this stuff, too. So that's called the tab transformer, one of the first ones to come out, a transformer for tabular data. And again, it's pretty simple. All you do is in any kind of input that you have, you will have some categorical variables like blood pressure, things like that. Not blood pressure. Bad example. Gender and so on and so forth.

And so what you do is you take all the categorical features, and for each categorical feature, you create embeddings because a categorical feature is just text. A categorical feature is just text, so you can create text embeddings for it, no problem. And you take all the continuous features, cholesterol and blood pressure and whatnot, to go to the heart disease example, and then you just correct them all and just create a vector out of them. It's just a vector.

Then you run the embeddings for all the categorical variables through a nice transformer block. And you can see here it's exactly the block we have seen before. No difference. And then at the very end, when it comes out of the transformer, you take all the contextual stuff coming out of the transformer, and then you concatenate it with the continuous features. And then you run it through maybe one or more dense layers and boom, output. So this is a tabular data transformer. And there are many refinements, improvements over the years that have come since then.

But the key thing I want you to remember here is that categorical variables can be very easily represented as embeddings. That's the key. So that's that. Now once the input has been transformed into this common language of embeddings, we can process them without changing the architecture of the block itself because all it wants is embeddings. It's like, you give me embeddings, I give you great contextual embeddings out, and nobody gets hurt. That is the deal with the transformer stack.

Since the transformer is agnostic to the kind of input, as long as it comes in as the form of an embedding, you can use it for multi-modal data very easily. So for example, let's say that you have a problem in which you have a picture that you have to be sent in, some text that goes in, a bunch of tabular data coming in. Well, you take the text and do language embeddings, like we know how to do. You take the image and do image embeddings, like we just saw with the vision transformer. You take tabular data under tabular data embeddings, like we saw with the tab transformer.

Once we do it, it's all a bunch of embeddings. And then you attach a little class token on top, send it through a bunch of transformer blocks, and then out comes the contextual class token, the contextual version. Run it through a ReLU, maybe a sigmoid or a softmax, predict the label, done. So this is extremely powerful, its ability to handle multi-modal data And that's why, for example, if you look at Google Gemini 1.5 Pro, GPT4 Vision and so on, you can send it images and a question, and you'll get an answer back. Because every modality that goes in is cast into embeddings.

And once it's embeddingized, then the transformer doesn't care. It'll just do its thing. It will decide, for example, that this word in your question actually is highly related to that patch in the picture. It'll just figure it out. OK, that's all I had because it's that time to break at 9:55. Perfect. All right, folks, thanks. Have a great rest of your week.

[APPLAUSE]