[SQUEAKING]

[RUSTLING]

[CLICKING]

PHILLIP ISOLA: OK. Welcome again, everyone. Sorry for the delay. So, yeah, we'll get started. Today is going to be about transformers. So hopefully you're all excited, because this is the architecture that you should use today.

Like, this will change. Next year, there'll be a new architecture. Don't worry, things will change. But this is the one that will get you the jobs today.

OK. This is going to be a little bit more of a practical lecture. With graph nets, we talked about the general class of all the functions that could represent, and so forth. And there's so many different possible ways of parameterizing graph nets. But with transformers, we're just going to say there's one concrete instantiation that works. And that's what we'll learn about today.

So we'll talk about three key ideas behind transformers. One is this idea of operating over tokens. The next, this is going to be the only thing that's actually new. It's going to be called attention. Everything else will actually be just new names for old ideas. And the last one we'll be talking more about, positional encoding, which we've seen a few times now. And then I'll show a few different architectures and some ways of using these things.

And so, as I said, there's so many architectures, but right now there's just one to rule them all. This is the one-ring architecture. Don't worry. It will change. This isn't going to be the final architecture, but this is the one right now.

So there's this interesting book called *Don Quixote,* written by Pierre Menard. And you probably are thinking I'm saying something wrong. It's written by Cervantes. But there's a short story that another guy, Borges, wrote about this guy, Pierre Menard, that rewrote the *Don Quixote,* word for word, exactly the same words, but with entirely new meaning.

And in a short story, the meaning is new because the context is different. It's a different historical era, and the life of Pierre is different than the life of Cervantes. And you're going to hear the same exact story. And if you read this, you would read the same exact story, but it'd have entirely new meaning, because the context is different.

And I think that's going to be a theme of the lecture today, and also the theme of this whole class. We'll keep on telling you the same things, maybe word for word the same things. But because you have new knowledge and new connections you can make, it'll have new meaning.

So this is like a few pages, short story. If you want to read it, it's kind of fun. It's called Pierre Menard, author of *The Quixote.*

So, everything old is new again. We're going to be riffing on the same themes. So let's go back to convolutional architectures. We're going to see a limitation of them.

So let's say that I want to ask, how many birds are in this image. We saw that you can do that with a CNN. You chop it up into pieces, you process with filters, you aggregate the results, you count the birds.

But what if I want to say, is the top right bird the same species as the bottom left bird? So can a CNN do this? Well, sure, if it's deep enough, if the receptive fields are big enough, it can do it.

But it might not be the best way of doing that. Because, remember, the CNN filters are all local. And local filters are not going to be the most effective at making assessments about relationships that are more global in nature or distant.

So CNNs, they're built around this idea of locality. They factorize the signal processing problem into little local pieces that get put together, step by step. So they have the strong locality bias.

And that can be a good thing. That's good for a lot of problems. If I want to understand something about the population in this room, I don't need to go and look at Mars. I can just locally look at what's going on in this room. Most signals have-- the things that matter to assessing them are all kind of local. There's a smoothness to the world. But that's not always a good idea. And so transformers try to get around that.

Here's just a diagram looking at the receptive fields, essentially, of a simple CNN. So this is a two-layer convolutional network. It's convolutional because every output neuron in this layer is going to be taking input from just a local patch of the first layer. So it's like a length 3 filter over the input. And if it's convolutional, the weights would be shared among all these filters.

But notice what happens. If I want to make some assessment about the relationship between the first input variable x1 and the seventh dimension of input X7. So on the second layer of the network, let's say this is the input, this is the first hidden layer, and this is the output.

On this layer, only these hashed nodes, these hashed neurons have some dependency on x7. And for x1, it's these two over here. And on the next layer, the receptive field expands out by one, because the filter is a length 3 filter. And so you get a receptive field that expands by one.

You can just trace the arrows back, see which neurons get some-- imagine some dye is dropped onto that neuron, and the dye bleeds out over the edges of this graph. Which ones is it going to hit? Which are in the support of this neuron here?

So it's just x1 is in the support of this neuron, this neuron and that neuron. And x7 is the support of the three neurons on the right. But if that's my output layer and I only have this depth 2 convolutional network, then no neuron in the output has gotten dye from both x1 and x7. So it's impossible for this architecture to have made a decision that depends on the joint configuration of x1 and x7. So I can always make a deeper CNN, but this dye just bleeds out slowly in convolutional networks.

So the idea of attention is to say, well, let's have an operation that allows us to look more globally. But the problem, there's another way of doing this. What is another way that we saw that you can have neural networks that process data globally? What is one of the architectures we saw that doesn't have this locality bias of CNNs? Anyone want to shout out?

AUDIENCE: Fully connected.

PHILLIP ISOLA: Fully connected network, just a vanilla MLP. And a vanilla MLP with fully connected layers, every single output neuron is connected to every input neuron. So information is aggregated immediately globally.

But CNN said no, no, we want to chop it up and factorize the problem. And then attention says, OK, no, we want to look globally, but we don't want to have every neuron connected to every neuron, because that would just be too much processing. That would mean that for a mapping from Rn to Rn, we'd have n squared edges and n squared parameters. And it would just be too compute heavy, too many parameters, too much to learn.

Attention says, well, we'll look globally, but we'll look sparsely globally. So when you get a new question, you'll just attend to the parts of the input signal that matter. So if I want to count how many birds, I'll attend to these locations. It's exactly coming from the idea of attention in the human brain or in your own life experience. You can attend to the things that are relevant to your problem.

If your problem changes, you'll attend to different things. So you'll attend to these two birds if the question is the top right bird the same as the bottom left bird. And if the question is, what is the color of the sky, you'll just attend to the sky region of the image.

So that's the intuitive idea of attention. It's meant to model what humans might be doing when they process images or other types of signals. And we're going to talk about mathematically how that can be done. So I'm going to go through these three architectural pieces of transformers.

Attention will be the second one. That's really the new one in transformers. And then tokens and positional encodes will be just the same as you've seen before.

So new idea-- not really new, but sort of new idea-- number 1 is tokens. So it's a new name for what we actually saw in the GNN lecture. So a token is a vector of neurons. This is how I'll define it for the purpose of our course.

So in the GNN lecture, we talked about node attribute vectors or node feature vectors. So every node has a vector associated with it. And we're going to give that just a new name, which is a token. So a token is a vector of neurons.

And the way to think of it is that it's like a little encapsulated factor in my computational graph. It's going to be a little packet of information that will be a little bundle of information that will all be processed in a modular fashion. So we'll have a set of tokens. And these are the units that we're operating over in transformers. So in graph nets, the tokens were the nodes of the graph or the feature vectors associated with the nodes.

So we can have an array of neurons, and that's what we operate over in MLPs and vanilla neural networks. But in transformers, we should always be thinking about the basic data structure as an array of tokens, as opposed to an array of neurons. So it's an array of vectors or array of little encapsulated bits of information.

You could also generalize the notion of token to be not a vector, but it's some array of structured information. In transformers, the structured information is just a vector, but there might be generalizations to this that come about later in 2025 for the next year.

So I do want to note that I'm using the word "token" in a way that is not completely standard. I'm using it in a way that shows up in various computer vision literature, but is a little bit less common in natural language processing. So in natural language processing, people mostly talk about tokens as being the discrete units of the vocabulary that we're modeling.

So it's like a token is the letter A, B, C. There's 26 tokens for the English alphabet. Or we could tokenize with a different kind of vocabulary. It could be that every English word is a separate token. So there's discrete units of the vocabulary that I'm operating over is what natural language people usually mean by the tokens. But it's a kind of weird definition.

I prefer to think of the tokens as the chunks of information that you are factorizing your problem into. It's a more general definition, and that's the one we're going to go with in this lecture. So token is a vector of neurons. And a set of tokens is how I will represent some data.

So just like I could have ConvNets that operate over an array of neurons, I could have convolution over tokens that operates over an array of tokens. That would just be like multi-channel convolution. So I can have different data structures built on top of tokens, where now tokens are the new units, the new encapsulated units of atomic information. In a neural net, we have scalar valued neurons. In a token neural net, we have vector valued units.

And we could have a set that's unstructured, unordered. And we could also have a set of tokens. And in transformer land, the main thing we operate over is actually sets, as opposed to arrays. But there's always variations on this theme that you can make up.

So let's use an image processing example. But I want to just make sure we keep in mind that transformers, maybe why they're so popular is because they're very agnostic to the domain. The recipe is once you turn your data into a set of tokens, then you just use a transformer. And this is one of the reasons they just dominate across so many different domains. And basically, all domains where deep learning is applied that I know of, I think transformers are currently the most popular way of doing it, or at least the most performant way of doing it.

So that's one of the powers is they're very agnostic to the input signal. Their previous methods, like CNNs, were much more tied to the locality properties of the signal. So how do you turn your data into tokens? And then we'll just process tokens. That's the general idea of transformers.

First, the domain expert turns data into tokens, and then generic transformer processes tokens. So for images, this is the standard way of doing it. But you can imagine a lot of other ways that one could do it. But you just break up the image into patches. Just like a ConvNet, you chop it up into patches, and then you turn each patch into a token by just projecting it into a flat vector.

And you could use some matrix that you multiply the pixel values by W tokenize to get a D dimensional vector. And that could be a learned matrix, or it could be somehow hard-coded. So just a linear operation.

And I've basically already said this-- but when operating over pixels, we're operating over scalar-valued measurements. What's the red color at that pixel? When operating over tokens, we're representing the input as an array of vector-valued measurements. And the vectors are just the projection of a set of RGB measurements in a patch onto a fixed dimensional vector in Rd. Question?

AUDIENCE: All the tokens are generally the same size?

PHILLIP ISOLA: Yeah, they do. So do all tokens generally have the same size? Yeah, usually they do. They're generically like a set of vectors in Rd. And that is an interesting thing to maybe do for a final project or a research project. I think variable size tokens could be really interesting.

It's not a standard way of doing it. Probably there are some challenges with hardware. Another idea of transformers is that they just match really, really well to the GPU hardware and the software that we use.

And if you have variable-size tokens, now there's something you have to keep track of. And how do how do you map that onto your hardware? It might be non-trivial. Question in the very back.

AUDIENCE: [INAUDIBLE]

PHILLIP ISOLA: Yeah, question. If you're tokenizing an image, do you want your patches to overlap? And the answer is in standard architectures, no, you generally don't overlap. But that might be a good idea. Just like in convolutional networks, you can set the stride that's a parameter. And you can have that same parameter here.

But the current standard way of doing it is no overlap, as far as I know. That might change. We'll hear one more, and then I'll move on.

AUDIENCE: So are the tokens just, like when you take the patches, the patches are also just pixels.

PHILLIP ISOLA: Yeah.

AUDIENCE: And it would just be the tokens would be the pixels in vector form.

PHILLIP ISOLA: Yeah. The question is, are the patches just pixels. Yes. So you flatten the patch into just a list of numbers. It's a long vector that is the length, depending on the size of the patch. And then you project that into a whatever dimensionality you want to process D dimensions.

So you can tokenize anything. And again, this is the power. The general strategy is you take your signal, your data you're processing, you chop it up into little chunks where the chunks make sense as the units of your domain. So usually that means local chunks, because locality is a very good inductive bias.

So in images, the standard thing is chop up into non-overlapping patches. In language, the standard thing is to chop up into little snippets of words which are called byte pairs. So I won't go into the details, but basically the kind of phonemes or the units of the language that you're operating it over, you could operate over single characters or three characters or entire words, but the one that people like is just called a byte pair. It's like two byte representations of the language.

So you can learn this for English. And there might be a byte pair that represents T-H, and there might be a byte pair that represents I-N-G. So that's how you carve up English and other natural language. And how you carve up sounds will often just be chop into chunks like you do for an image.

So once you have done that, then everything else is domain agnostic. So this is the only part where you have to actually consider your input domain. I mean, yes, when you think about the loss function and other formulations, yes, you think about your input domain. But this is the main part.

So here's a notation that we're going to use. So we're going to think of the tokens as just vectors. And I'm often going to describe an array of tokens or a set of tokens, because it will turn out that actually the ordering is going to not matter. We'll have architectures that are permutation invariant over the ordering of the rows of that matrix as this matrix T.

So I'm going to just transpose my token vectors. Now they're going to be row vectors. I'm going to stack them up. And that will be this matrix T that will simplify a lot of the math.

So when we're operating over tokens, this is the notation we'll use. We have n tokens in the rows and d channels. So we're going to use the same dimensionality for all tokens to answer that question. And if you've tried to use different dimensionality, the notation will become more complex. And you'd have to do some more bookkeeping. And yeah, it would be harder.

So just like with regular old MLPs in networks built out of tokens, we have two basic operations. One is a linear combination operation. And the other is a unit-wise nonlinearity.

So what is a linear combination operation? It's just a generalization of the linear combination over neurons. So here's a linear combination over neurons. This is a linear layer. I have x in is a set of scalars-- x1, x2, x3. And x out is just another scalar. And I have a weighted combination, w1 times x1, and so forth. This is the regular old linear layer going from R3 to R1.

So what is a linear combination of tokens look like? Oh, and right, we just can write that in matrix form as w times x in. You've done that a lot on the P sets and in other places.

A linear combination of tokens is just going to be a weighted combination of vectors. So I'll have vectors with D dimensions. But if I only have three tokens, three such vectors, I'll have three weights that I can learn or decide to set the values of to create a linear combination. So it's going to be a weight times the first vector, plus a weight times the second vector, plus a weight times the third vector.

So you can represent that, again, in matrix notation, if we're using our T where T is n rows by d columns, where d is the dimensionality of each token vector. Then w times that matrix T is just like taking a linear combination of the rows.

So you can already see here that the token linear combination is less expressive than if I did the same thing over all the individual neurons that make up those tokens. It's like a low rank transformation over the neurons, because I only have three weights that determine that combination, but I actually have 12 neurons that make up all of those tokens. And so you normally would have 12 weights if it were just a linear combination of neurons.

So this is a less expressive family of layers. In fact, it's a low rank operation. But we're thinking of the tokens as these bundles, these encapsulated units of operations. We've just now defined how to take linear combinations of those encapsulated units.

So the next thing we need is the equivalent of a ReLU, the point-wise nonlinearity. And with tokens, we can have a token-wise nonlinearity. Instead of being a neuron-wise nonlinearity, it will be a nonlinearity that operates on each token independently and identically.

So here's my point-wise nonlinearity from MLPs. We have x out is equal to, for each dimension of x in, we just take a ReLU of that dimension. And a token-wise nonlinearity is just going to be, for each token in my array of tokens, I will pass it through a nonlinear function F. And it will be the same nonlinear function applied to every single token.

So typically in transformers, F is going to be itself an MLP. So transformers are like a meta architecture. Like, the units of the architecture are other neural networks.

And you can also think of this operation as equivalent to a convolutional filter that is run across the sequence of tokens. It's a 1 by 1 convolutional filter. So what does that look like? It looks like this.

I have a sequence of tokens. Now think of this as like a spatial sequence or temporal sequence. And the channels are in this token dimension. The vector dimensions of the token.

And the convolutional filter will just be something that operates on each token independently and identically. So this is what the token-wise nonlinearity looks like. And you can also now map that in your brain onto a point-wise nonlinearity in an MLP, that would just be like I have vectors of length 1. And my operation them sliding across that sequence is a ReLU or a tanh or sigmoid, some point-wise nonlinearity.

So now we've basically built what I'll call token nets. Another name for this would be graph nets. But I'm going to call it token nets for now, which is almost identical looking to a MLP.

So MLP is linear combination of scalar-valued neurons, point-wise nonlinearity, and so on. And token net is linear combination of tokens. Take a weighted sum of these three items, followed by token-wise nonlinearity, so each arrow on the second row is itself parameterized by an MLP-- it's a complicated nonlinear function-- and then repeat that process.

So the motif is the same. It's just over these more complicated units. As opposed to being scalar-valued units, which are called neurons, it's vector-valued units, which I'm calling tokens.

So I already mentioned, but this is also equivalent to that unrolled GNN that I talked about a few lectures ago. So an unrolled GNN, we have this aggregate function, which is doing some kind of weighted combination. And then we have a combined function, which now you can understand to be a unit-wise, a node-wise nonlinearity. And then we repeat that process.

And GNN is also often have the property that you share weights across depth. You don't have to, but you can. Whereas transformers typically have the property that you don't share weights across depth, but you can. So this is a choice. With the standard transformer, you have separate set of weights here and on the next layer. And I'll tell you how those weights actually get specified in a minute.

And just to remind you, here was that picture of the unrolled GNN. So an unrolled GNN, we have our graph. And we take these-- now I'm calling them tokens, they used to be called the node vectors-- and we aggregate. But this is going to be in a critical difference.

In the GNN, how we aggregate is determined by the graph connectivity. It's the adjacency matrix of the graph that tells us how we do this aggregate, where the arrows are in that linear combination. And additionally in the graph, an aggregate might be nonlinear. It doesn't have to just be a linear combination.

And then we also have a particular way of combining that through an update step. But it looks very much like a transformer that I'm showing you now, the token net version of the transformer, except that these arrows are not fully connected. They are determined by the graph structure.

So a transformer is going to be like a graph net, but it's a fully connected graph net. So we have all the arrows at every single aggregation layer. And I'll tell you how those arrows' values get determined, which is going to be called attention. But you can think of it as a graph net over a fully connected graph, typically without weight-sharing in depth. Any questions about the relationship between these different neural networks-- MLPs, CNNs, graph nets, and now this token net I'm describing?

AUDIENCE: Can you talk more about the token-wise nonlinearity and how it's changing?

PHILLIP ISOLA: So the token-wise nonlinearity will be typically an MLP, and the parameters will be learned with backpropagation. So the free parameters are learned via optimization.

AUDIENCE: And within there in that MLP, there's point-wise nonlinearities in that as well.

PHILLIP ISOLA: Yes. So it's like this network of subnetworks. And in the MLP, there are also point-wise nonlinearities. And one more?

AUDIENCE: The aggregate functions are then at least the equivalent of the activation functions on the transformers then?

PHILLIP ISOLA: The question is, in GNNs the aggregate function is learned-- or it's parameterized, and it can be learned. And in transformers, I didn't tell you how we're going to learn these weights. In transformers, that will be done via this thing called attention, which we'll get to in a minute.

And the token-wise nonlinearity in an MLP is not learned. The point-wise nonlinearity is not learned, typically. But in a transformer, it typically is learned. It's a parameterized MLP. And in a graph net, I think it would also be learned. It's this update function, which typically has learnable parameters. OK.

And don't worry too much about the semantics here. Like, it doesn't matter if you call something a graph net or a transformer. You should understand the concepts and how there's these particular building blocks that can be combined in different ways to achieve different effects. And there's trade-offs. And if you understand that deeply, then the names are just something that is secondary to that.

So that was the first new idea was we're going to operate over tokens, which are vector-valued units, instead of neurons, which are scalar-valued units. The next idea, which is going to actually be technically new, is going to be attention, which is how do we take the linear combination exactly between tokens?

Let's say I have a token net. What I could do is I could just learn the weights of the linear combination. Just like in an MLP, I learned the weights of the linear combination.

So free parameters I'll draw in blue. And this could just be a weight matrix, which is going to be three numbers, in this case, because I have three tokens. So it would be a three by one weight matrix and one output. Three inputs, one output. And W will be free parameters that are learnable via backprop. That's one option. You can do this. That will work

But transformers actually are coupled with this other idea called attention. And in attention, the weights that you use to do this linear combination, now they're not free parameters. So I'm not going to color them in blue. I'm coloring them in red. And if you remember in some of the readings, red was our color for activations.

It turns out the weights I used to combine the tokens with the linear combination are going to be coming from activations elsewhere in the network. So A is going to not be free parameters. It's going to be a function of your data that you're processing.

So there's a few different ways you can do that. We'll go into the full details. But just think that there's going to be some function F that takes in your input data you're processing and determines this linear combination matrix A. So A is going to be set by some function, which I'll describe, of the data. It's not going to be free parameters. And T out will be just a linear combination of T in, according to A.

So let's get some intuition for what this attention matrix might be doing, this attention matrix A. So here's an image. And let's say I want to process the query, how many animals are in this photo? So I told you the intuition is I should look at the three animals and then count them up. And that's exactly what we'll do.

So I will attend to-- remember, tokens are representing patches of the image, at least on the first layer of processing. They're representing patches. So what are the patches I will attend to in this image to answer this question? Should be any patch that has an animal. And it's a little hard to see, but there's a giraffe head in this one here. So this is zebra, impala, and two giraffes.

Now, every token is a vector of numbers that are descriptive of what is going on. There's some representation of that part of the world. So at the first layer, it'll just be the concatenation or the flattened list of all the RGB values. But later on, it could be more abstract.

And here I'm just indicating that there might be one element of the vector which is an indicator function. Does this patch contain an animal or not, or maybe the head of an animal or not? Because each animal has one head.

And then what I can do is my attention can be allocating one unit of attention to each of these tokens, and zero units to any other token. And then if I take a linear combination with weights of 1 for each of those tokens and 0 for all the other tokens, that will achieve the correct answer of outputting a new token vector whose value is 4 in the dimension that counts animals.

So I've just kind of hand constructed here a transformer, one layer transformer, that counts the number of animals in the image by simply attending to animal heads and then taking a sum over all the things I attended to where the attribute vector of each thing I'm attending to. The token vector is just a 1 if there's an animal head, or a 0 otherwise. Or in fact, in this case, it could be a 1 no matter what's in the photo, because the attention already only selected to look at the animal heads.

Here's another example. What is the color of the impala? So what should I attend to here? I should attend to these tokens, these patches. And now I have another dimension of my token vector which maybe represents the average color within that patch.

And if I take the average over those, I take a weighted sum over the things I'm attending to, I will output the average color of the impala. And that will be maybe a good ensembled response about the answer to this question, what color is the impala? So this is just the intuition, but hand constructing how these operations could actually answer queries of interest. Question?

AUDIENCE: How do you interpret the [INAUDIBLE] of the T out? Is there a [INAUDIBLE]?

PHILLIP ISOLA: T out here is meant to be-- oh, yeah, I suppose that maybe I should use a capital letter, but just one token output. It's like I'm just outputting one single token to answer my question in this example. So yeah, the more general form is I'll have an array of tokens represented by the matrix T, where the tokens are in the rows of that array.

So here is going to be the mathematical formulation of the most common kind of attention. It's called query-key-value attention. And remember, the transformer is mapping from a set of tokens to a set of tokens, so T in to T out. And this will be one layer of the transformer, one attention layer, so one linear combination.

So how is it going to work? So I will have a question. And then I'm going to use the question to determine the attention. So the question, the way I determine the attention is like that function F that produces A.

So here's what it looks like. So my input signal is a set of tokens. So it's actually just going to be thought of as an unordered set. Just here's a bunch of different patches in the image. And every token has this vector associated with it.

And the question will also be represented as a token, which has a vector associated with it. And I'll have a special operation called query, which will be to emit some vector based on the contents of my question's embedding vector. So the question the question is a token, which is a vector. And then I can transform that to emit some kind of query. So that will just be a linear transformation of that input vector.

And each token in my signal that I'm processing will have a function called key, which emits something that the query is meant to try to match against, to see if the query matches the key. So tokens represented by a vector, and they can emit either queries in order to say what should I attend to, or keys to say what is my content that my query should match to. And we will check whether a query matches a key via a dot product between the key vector and the query vector.

So this that horizontal line is the key vector. And that vertical yellow line is the query vector. And this is just meant to be the dot product, like the matrix product, of the 1 by D vector times the D by 1 vector.

So what will happen here? Well, my query is relevant to animal heads. What is the color of the impala's head? So my key will match that query if I'm looking at an animal head. So the giraffe will be 0.2, but it will match even better if I'm looking at the impala head, because I'm specifying the impala. So that would be, if I train this thing to be a reasonable system, then this would be a reasonable set of queries and keys.

And the similarity between the query and key is what I will attend to. So I'll get this kind of score vector s over here on the left. And this will just be the dot product between the question's query vector and each token in the signal and processing's key vector.

Then I will also have one more operation called value, which will be another transformation of my token vector, into what I'm going to report to the next layer of my network. This is an analogy from databases where I can ask a question. Each bin in my big database-- it can know what it contains. It can say, here's the key, here's the relevant information I contain. And then when the query matches the key, you open that part of the database, that bin in the filing cabinet, and you pull out the values and get some information back.

So now the operation will be I take the similarity vector s. That's going to be the weights of my linear combination, but the linear combination will be applied over the value vectors for each token. So T out will be a weighted combination of the value vectors of each token. So those weights alpha, or A1 through An, over here, will be determined by the score. And it'll be determined by the similarity vector by taking a softmax, just normalizing that vector so it sums to 1.

So I'm going to then get finally my A's which are the elements of my attention matrix, which tells me how much attention to apply to each of the tokens' values. And then I'll just take the weighted sum, the linear combination.

You have a bunch of questions. So let me quickly finish this slide, and then we'll go over it.

Like, this is just a mechanism. This is a concrete mechanism that ends up working really well. So it takes a while to get comfortable with it.

But I just want to point out that the key query and value vectors are just linear projections of the token vector. So we have one matrix that's learnable, that transforms the token vector into a query; another learnable matrix that transforms it into a key; and another one that transforms it into a value. So these are the learnable parameters. These projections are the learnable parameters of an attention layer.

Yeah, a few questions. Let's go here.

AUDIENCE: So is the tokenizer the same for-- so is it a multimodal function in that both text and image can be tokenized the same way?

PHILLIP ISOLA: Yeah, so can text and images be tokenized the same way? So no, you'd have a tokenizer for text and a tokenizer for images. And this is after you've done that. Now I'm just kind of abstractly saying there is a token that represents that question and a token that represents each patch. And here's how you will use that question to allocate attention to those patches, and then process the image.

Question over here?

AUDIENCE: I was wondering. So the query image is, for example, taken in the day. But the key images are taken at night. At what point during the layers do we modify that? So we're pairing an image taken in the morning compared to image taken at night. Same object, but how do we change that with respect to the color?

PHILLIP ISOLA: So the question is, let's say the queries are coming from an image taken during the day, and we're processing an image taken at night. So the keys are taken from an image taken at night. How do I bridge that gap? So learning will have to solve that for you. So you'll train on being able to query from daytime to ask questions about nighttime. And the parameters will have to adjust in order to be invariant, maybe to the time of day or appropriately answer that question.

AUDIENCE: Do we choose the A, the softmax value?

PHILLIP ISOLA: So all the learnable parameters of a transformer can contribute to this. In the attention layer, there's only three matrices that are learnable, which are the projection from the token vector to the query vector, the projection from the token vector to the key, and the value.

Question in the back? Over here, yeah.

AUDIENCE: So for something like a segmentation model where the query basically does not change, what does the query [INAUDIBLE]?

PHILLIP ISOLA: Great. So, what if the question doesn't change? I'm just trying to solve one task. I'm going to talk about self-attention next, which will address that setting.

Let's do one more over here.

AUDIENCE: What's the meaning of the dot product between the key vector and the query vector you have here on the illustration? Do they have any significance?

PHILLIP ISOLA: The dot product between the key vector and the query vector is meant to say how relevant is this key to this query. And so dot product is like measuring the similarity between those two vectors in some way. And an appropriate learning system would arrive at projections W, K, V, and Q that make it so that when the text is about an impala's head, then the query vector will have values that are similar to the key vector values for images of impala heads.

AUDIENCE: Right. So particularly interesting for the first one is value 1. Does that indicate a perfect match?

PHILLIP ISOLA: Oh. That looks like a typo. Sorry. So it's 0.1 up here. And I think I forgot to make it 0.1 down there. So that's just a typo. Yeah, good question.

OK, let's do just one more.

AUDIENCE: There seem to be a mapping from the question, what color is the impala head, to the query vector. So what do you mean that mapping? How we can [INAUDIBLE]?

PHILLIP ISOLA: That mapping, this query function that maps from the question vector, so the token vector that represents the question, is given by this equation up here, which is just a linear projection of the token vector. So it's just a learnable matrix, WQ.

But it could be other functions. I'm just telling you that that's the standard one in transformers. But you can always change that. It could be an MLP.

So let's move on. Now, I'll also tell you that the problem set coming out next week, you're going to implement a bunch of transformers and all the variations. It'll implement GPT. Like, you'll intimately know how transformers work. It's a little bit more of a practical problem set.

But I want to now talk about the next idea, which is self-attention. So self-attention is saying, rather than having my queries come from some external question, like some text, my queries could come from this signal itself. So I could have it that every single token in the set of tokens I'm processing can emit queries to the other tokens. And that's called self-attention.

Here's just an intuitive example. If this patch wants to get a better estimate of what it is, maybe it should attend to other things that are similar, and then ensemble or average the results to now say, I'm not sure if I'm an impala or I'm a gazelle or what I am. So I will attend to other things that look like me in the image. And then I will update my own representation by aggregating information from the other things that are similar. That's an intuition for one way that this could solve problems of interest. But in general, it might be learning very complicated ways of processing information in the signal.

Let me show you this video of the self-attention applied to patches in these videos when you are looking at the self-attention of a token that corresponds to a place on this horse to every other token in the image. And for the dog, it will be a token on the dog, and how much self-attention it applies to every other token in the image or in the video.

So what you'll notice is that the horse token attends to the horse patches. So it's solving the segmentation problem. It's like, if I want to understand something about this part of the image, I should attend to other things that are related to it, and I shouldn't cross object boundaries. This is the natural segmentation of that scene. And that's what the attention function learns for this transformer, which is doing visual recognition.

Yeah, question.

AUDIENCE: So in the examples that you're showing, it's like the patch is able to intelligently figure out which parts of the image to attend to. But given some arbitrary question or arbitrary task, how does it choose of all the tokens in the image to say, like, these are the ones I care about?

PHILLIP ISOLA: So how does the attention mechanism figure out what to actually attend to, what its query vector should be, what its value vector should be? That's just machine learning. That's just backpropagation to minimize your objective is going to find the setting of all the weights, the mechanism that will achieve good behavior. And it turns out the mechanism that achieves good behavior often has an intuitive interpretation like I'm giving you, but it doesn't have to.

To solve recognition, maybe I don't have to have the patches on the horse attend to other patches on the horse. Maybe I could have attended to the background and done some inference about the context. If it's on the field, and therefore it might be a horse. But it turns out that it comes up with a more intuitive strategy, which is the horse attends to other horse parts, and then decides what it is.

Yeah.

AUDIENCE: It's more of a similar question, but this seems like an observation, but does it actually always happen? Or is there a way to interpret or force the model to train this kind of behavior?

PHILLIP ISOLA: Yeah, great question. This seems like an observation. Does it really happen? Can you enforce this behavior?

This is more empirical science. This is saying, we have an architecture and we're investigating what it discovered. And then I'm trying to provide intuitions for why it would work that way, why it discovered that solution.

But it's not provable. It's a property of the data and the statistics of the world in combination with the optimization and the transformer architecture that ended up here. And there's a lot of interesting empirical science on what do transformers learn, what internal representations do they have, what are the circuits like. There's this idea of interpretability research, trying to understand what these things learn.

But there's another lens, which is just who cares what they learn-- let's understand ML theory and how gradient descent on objective functions with high-capacity networks will converge on good solutions, but not care about the internal structure. So you can look at it from either lens. I think I need to move on, but we'll come back and have more time for questions soon.

So here is a fully connected layer with learnable parameters for how to take a weighted combination of the input token sequence to the output token sequence. And here is the idea of self-attention, which is that the weights of that linear combination are a function of the input sequence. So the F that's going to determine our queries and keys, or in particular our queries, is going to be coming from the input token sequence T in.

So how is it going to come from them? It's going to be exactly the same type of math that I showed you a few slides back. But here's how it looks for self-attention.

So I have my T in. And every single token in my input sequence is going to emit a key, a value, and a query vector. It's going to emit those via WQ, which is a projection from its token vector into a query vector WK, which is a projection from its token vectors to its key vector, and the same for the value vector.

So now I will get a set of queries, a set of keys, and a set of values. I can stack them up into the query matrix, which is for every row in that matrix it will be the query for one of my input tokens, for every key matrix and the value matrix. And this is just saying that I can represent the query matrix as just the matrix product of-- it's just a matmul, it's just a matrix multiply of the token matrix times the learnable projection to queries.

So this is another nice thing about the design of transformers is everything just becomes these matrix multiplies in a very simple, notational form. So this is my queries. And they're just T in times WQ.

So now I take all of these matrices. And in order to get the weight in the linear combination for this unit of T out from this unit of T in-- so remember, the linear combination is over the values. The values are in this teal color. So I'm going to take a linear combination over the values. And the weight that I'll assign to that linear combination is going to be the inner product, the dot product, between the key and the query for this token

So the amount of self-attention that I apply to myself is going to be given by this gray box, which is just this dot product. And if you work out the linear algebra, you can realize that this will just be the matrix multiply of the query times the key matrix. And then that will just be every query vector gets multiplied by every key vector. And that results in the value.

And the values will then be stacked up into an n by n matrix. I'll have n input tokens to n output tokens. That's an n by n set of weights that determine that linear combination.

So mathematically, we can condense all of this into this simple equation. This is the famous attention equation, which is going to be the query matrix times the key matrix transpose. So if I transpose, I'm going to be taking the dot product between all the pairs of queries and keys.

And then I'll divide by the square root of the dimensionality of the vectors. So the keys' queries and values are M dimensional vectors. They don't have to be the same. They don't have to be D dimensional.

Remember, the tokens were D dimensional, but in general the queries' keys and values could be of some other dimensionality. I'll divide by that dimensionality. And then I'll take a softmax to ensure that everything sums to 1. So that has some nice normalization and numerical advantages.

And then finally, the output will just be the attention matrix times the value matrix of the input set of tokens. So it's a lot of little nitty-gritty linear algebra to work out. But it comes to a very simple equation in the end. Yeah.

AUDIENCE: Like, is there a good intuitive reason for why we bother learning parameters or matrices for human case separately, rather than learning the values of the A matrix directly? At the end of the day, T in is going to be a matrix multiply, which is a function of two learned matrices, which is A. Dot product with another learned matrix, which is [INAUDIBLE].

PHILLIP ISOLA: Right.

AUDIENCE: So is there something about decomposing A into the product of Q and K that leads to like-- I'm not asking for a theoretical argument.

PHILLIP ISOLA: Yeah, no, no, it's a good question. Like, why not learn A directly? The basic statement is empirically the transformer recipe-- I'll specify it-- just seems to work the best at so many different things. I don't know about direct comparisons between learning A versus using this outer product of the queries and the keys that are also learnable projections. But you can start to count how many learnable parameters there are and what is.

So at the end of the day, we have some parameterized mapping from T in to T out. And you can think, if I learned A directly, I would have to learn n squared parameters, where n is the number of tokens in my sequence. If I learn WQ, WK, and WV, I'm also learning a lot of parameters, but it's not dependent on the sequence length. Now it's dependent on the dimensionality of my token vectors. So it's dependent on D, as opposed to n.

Now transformers are typically applied in modern AI to extremely long sequences of tokens, maybe like length million sequences of tokens, even. Like, modern LLMs, you can have paragraphs and paragraphs and whole books that you're processing. So n is usually very large, and D is much smaller. So there's actually fewer learnable parameters. That's just one perspective.

OK, one more.

AUDIENCE: Yeah, I just wonder if the size of the I or n or a different tree or different update for your neural network could be variable or has to be fixed? Because I feel like language is so different from images. And the sequence, the length of your words, can change different [? things. ?]

PHILLIP ISOLA: Yeah, OK, I'm not sure I've got the first part of the question quite. But I think you're asking, shouldn't we have a different way of processing language and vision? Can you adapt your architecture to that?

And the answer is, yeah, that is of interest. But again, the power of the transformer and the transformer paradigm is only put in domain knowledge into the first step, plus a few other places, like in positional encoding, which we'll get to, you can put in domain knowledge. And otherwise use a very generic computational framework, which has the advantage of just if you homogenize, then you can get advantages of everything runs on the same commodity hardware. The code bases are all the same. The lessons are transferable between different modalities. So there's a lot of advantages to that.

But specialization to modality is of interest as well. It's just a trade-off, a back and forth. How much do you specialize versus how much do you make homogeneous approaches? I'm not sure I quite answered the question, but I think I need to move on to get through everything. We can talk after.

So as I said, the attention layer is, at the end of the day, just a linear combination. So we can now think of a family of different linear combinations of neurons that we've seen. So the first one was in the MLP, the fully connected layer, where every single edge in the mapping from input vector to output vector is a learnable parameter. So we have n squared learnable parameters. If the input is in Rn and the output is in Rn. And we also have some bias terms that we can add, n bias terms.

In convolution, we have a much lower rank matrix. But remember, we said that the convolution can be represented as a matrix. For a fixed dimensional input, we can represent the convolution as a fixed dimensional matrix.

But the interesting thing is it has this Toeplitz structure, where the diagonals all share the same value. So the colors indicate the unique values in this matrix. So there's far fewer learnable parameters. There's only four learnable parameters if we include the bias, as opposed to n squared learnable parameters, for this toy problem where the convolutional kernel size is 3. So every three neurons in the input map to one neuron in the output.

And convolution has this nice property of translation equivariance, conv of trans x is trans of conv x. And that comes because of this weight sharing. You kind of see it. Like, if I translate the input, I'm just the same values at a different location in that matrix.

So what is the transformer attention matrix look like with this notation? So what do you think? This is probably pretty tricky to work out just in your head. But what do you think it's going to look like? I'm going to draw it on the next animation. So anyone want to guess? Like, what is the transformer's matrix of unique values going to look like?

OK, yeah, let's go here.

AUDIENCE: So it's not going to necessarily look sparse, but it should be low rank.

PHILLIP ISOLA: So it will be low rank. You said it's not going to look sparse. I think it does look sparse, but we'll see, we'll see. Yeah, OK, any others? It's going to be some low-rank sparse matrix. It's a little hard to work it out.

So I also have to tell you how I'm notating things. So I mean, it's not like you were wrong. So I'm going to notate things where I'm going to take the first three neurons and call that one token, and the next three neurons another token. So I'm factorizing the input signal into these two tokens now.

And then those two tokens get to attend and interact with each other. But that just corresponds to using the same set of weights for each element of the token vector, because a linear combination of rows of a matrix is using the same weights for every element of the row when I'm taking that combination.

So you can work it out. Think of it for yourself, but this is what it looks like. So transformers have these layers or linear layers. Of course, the weights are coming from queries and values. So there's another mechanism on the side. But at the end of the day, it's just like this low-rank sparse transformation. It has fewer linear learnable parameters, which is advantageous in some ways.

And then the other interesting property, which I'll describe in a minute, is that transformers are equivariant with permutations of the input. But we'll come back to that, just like graph nets.

OK. So here's the MLP. Here's the vanilla transformer with self-attention. And this is roughly the standard architecture we use in computer vision. The architecture we use in language processing has one small difference, which I'll come to in a minute.

A few little details to add on top of this-- I think I won't talk about multi-headed self-attention. It's in the reading. There's a section on it. But the idea is that, rather than just having one attention layer, I can have k attention layers in parallel, and then I can aggregate them. And each attention layer can be attending to different things. One can learn to attend to shapes and one can learn to attend to textures, for example.

OK, so here is the complete vision transformer architecture. This is the standard way of processing spatial signals. So images in particular, but also audio and other things can be used with the same architecture. And there's only a few things that you haven't seen. So it's going to be a set of tokens, multi-headed self-attention, which is MSA. So that's just multiple runs of attention, then combining them all together, and then point-wise nonlinearity, which is going to be an MLP.

The blue are the learnable parameters. Everything else is not learnable. So the queries and the keys and the values, which are indicated by this F, well, that's really the queries, but there's also keys and values. That's learnable. And the parameters of the token-wise MLP are learnable. Everything else is not.

These pluses here are residual connections. So I think in the CNN lecture, we talked about ResNets. So we'll just take an identity pass around all this processing. And that has some advantages.

And then there's one other little layer, which I'm labeling token norm. It's often more commonly called in the literature LayerNorm. There's not really an obvious kind of layer structure here. Really, this operation is just taking the vector and normalizing it to have zero mean and unit variance. So I'm taking the elements of this vector and I'm just dividing by the variance and subtracting out the mean.

So that connects a little bit to what Jeremy was talking about and what you've done on your problem sets or what you're working on your problem sets, where it's good to normalize activations and weight updates in your network for a lot of reasons. One reason that Jeremy alluded to is, for stability of optimization and taking the steepest descent direction, you need to think about the norms. And you might want to normalize your weight updates in a particular norm. And your problem set goes into that.

Here we're not normalizing the weight updates. We're normalizing the activations. But you can understand that normalizing activations will have a consequence on changing the size of the weight updates. Because if I backprop through normalizing the activations, you can see that the weight update will be a function of the scale of the activations.

So LayerNorm has desirable optimization properties. Exactly what those are is basically open science. And it's not quite clear why LayerNorm is right thing to do, but it's what people do.

So you'll implement transformers on your problem set, but I want to quickly walk through a kind of pseudocode of it just to show you how simple these things are. They're so easy to implement. That's one of the reasons why people like them.

So here is a transformer in the vision transformer style that is processing a set of tokens T. So we first tokenize the input. That might be domain-specific knowledge, like break up your image into patches.

And then for each layer, we're just going to run the same operation. We might have different weights per layer, but otherwise the same operation. We will take the matrix multiply of the LayerNorm of the tokens, times like this outer product of the key query and value matrices with the token matrix. And we're LayerNorming it. Then we get key query and value matrices.

And then we'll just take the matrix multiply of Q times K transpose, divided by the dimensionality, which here I'm noting as D, the dimensionality of the key query and value vectors. And then I'll add residual connections, and take the linear combination, other matmul. So everything is just matmuls and a few other simple operators. And this maps wonderfully onto modern compute that loves matrix multiplies. OK, so really simple. And that should work.

So there's one more new idea I have to get to before the end of the lecture. Well, there's a few more things to get to, but one more big new idea, which is positional encodings. It's not a new idea. Again, it's been seen in a lot of older architectures, but it was one of the key things that made transformers stand out.

So the first thing to know is that transformers are permutation equivariant, just like graph nets. So if I permute the input sequence of tokens, then you can work out for yourself that if I permute it, well, it's like taking T2 into the place of T1, then the output will also change the order. So the output representation of T2 will now be in this vector, as opposed to in the second vector. So permuting the input permutes the output

So transformers are essentially a set-to-set mapping. An unordered set maps to an unordered set. The ordering of the tokens doesn't matter.

So you can see this, because point-wise, token-wise nonlinearity F, it just applies to every token independently and identically. So of course, if I change the order of the tokens, it doesn't change anything. I'm just treating them all independently. And you can work out for yourself that attention is permutation invariant.

Because again, attention is, like, every pair of tokens, how much they attend to each other will just be determined based on the values in the token vectors, not the order. So because transformers are just point-wise, token-wise nonlinearity and attention, the whole thing is permutation equivariant. So transformer permute the input is equal to permute the transformer of the input.

So now, positional codes-- well, remember, if we want convolutional networks to not be translation equivariant, we tell the filter where I am in the image. And then it can make a different decision based on where I am. It's no longer translation equivariant. We saw that a few times.

So now you can do exactly the same thing with transformers. If I don't want it to be permutation equivariant, which oftentimes you actually don't. Oftentimes, the order matters. If I'm processing a sentence, the earlier sentences have causal impact. The earlier words in the sentence are causally determining the next words, but there's not quite the anti-causal direction. The statistics are different. It's not like a reversible sequence where permutation invariant statistical process.

So anyway, I can concatenate each token with a value that tells me where it came from in the input signal. What position was it in the input sentence? What location was it in the input image?

And here's how you do that typically. Not typically-- but this is the vanilla way of doing that. There's a lot of more advanced ways of doing it. But if I want to tell this patch here on the giraffe's head what position it's at, I could encode the xy location in Cartesian coordinates of that patch, but I'll typically represent this on a Fourier basis where I will say, what is the value of a sine function at that location? And I'll do that for different sines and cosines, or in the vertical and the horizontal direction. And this is just like an encoding of the position, but in terms of a different basis as opposed to being just like the Cartesian grid.

So these values tell me where I am in the image. The book chapter goes into some intuition about why Fourier positional encoding is advantageous over other types. But this is also an open science question. This one works pretty well.

And for whatever signal you have, there's a lot of domain expertise that can come into this part of the problem. How you define the positional encodings. Because that's the part where you tell the system what it means to be local for your domain. You give it the inductive bias of what locality actually represents. So for processing data on a globe, I can do positional encoding that's like latitude and longitude, but maybe on some kind of Fourier basis over spherical harmonics, over sinusoids on the sphere.

For graphs, one of the typical positional encodings is onto the eigenbasis of the graph Laplacian. So we mentioned this a little bit in the GNN lecture. But basically, I can take a graph and I can compute something called the graph Laplacian. And I can look at the eigenvectors of that, this thing that tells me location in the graph, canonical location in the graph.

And the coordinates, some of these eigenvectors of this graph Laplacian are described by these colors on the nodes. And so the first eigenvector is this one. It's really smooth. It's like, where am I globally? And then other eigenvectors are, like, high frequency, almost like a Fourier basis.

This is all domain knowledge. It's not stuff that you need to know intimately, unless you're working on that domain. But this is just to say you can introduce a lot of domain knowledge into positional encodings.

So those are the three pieces for the transformer. But there's one last thing, which is a lot of you will have seen large language models and already heard about transformers in the context of language modeling. So I want to tell you how transformers map onto language models, because there's one extra piece that is important. And it's called causal attention.

So first, we're going to come back to autoregressive models and generative models of language and other signals a little later. And one of the kinds will be an autoregressive model. But I'll tell you briefly what it is now, because it's extremely simple. And a lot of you already know this, because it's all over the media. This is how ChatGPT works, and so forth.

You take a sequence of words and you simply try to predict what is the next word in that sequence. So "once upon a time," and then you do the same thing autoregressively, which means you just take your prediction and concatenate it or post-pend the input sequence with your prediction, and then keep on going. Now predict the next word and the next word.

And you can train such a model, because you will train the predictions to match the data set of actual text that you observe from the training set. So we'll say, "once upon a time," and you can watch Claude or Llama, or whatever your favorite LLM, you can watch it write out things. It's doing exactly this.

We'll talk more about those later. And you don't have to do autoregression in time's arrow order. You can do it in any order you want. But that's just a detail.

So as I said, you have a training set, which is sequences and completions to the sequence. This is just text online for language models. And you are then going to use supervised learning to try to classify what is the next word in a vocabulary of possible words.

And then at test time when you're going to sample new sentences, you'll just say, "colorless green ideas sleep," it will predict, based on my training data, what might have come next. And maybe "furiously" came next. It's a quote from Noem Chomsky.

So that's how GPT works. GPT is Generative Pre-trained Transformer. So it's a transformer. And it's an autoregressive transformer.

So here's my normal transformer that I'm drawing, where it's self-attention. So the input sequence of tokens that represent the words, or maybe subparts of the words, get to choose which of them to attend to, update their representations, finally at the very end come up with a prediction for the next item in the sequence. And every time I run this, I will get a prediction of the next item in the sequence, and I'll shift the input over by one and repeat the process.

So there's a little issue here, which is that if I want to train such a system, I want it to be the case that my prediction for the word "furiously" only depends on the previous items in the sequence. Now with this architecture I'm showing here, the input has the word "furiously." And it could just learn an identity connection to say, oh, I'm predicting "furiously." And then my input already told me what the answer was. That'd be wrong.

If I'm predicting the word "ideas," I need to only be able to look at the previous items in the sequence. I can't look into the future if I want to be able to write a sentence in time's order. So what we do is we do what's called masking, causal masking of our attention matrix.

And that just means that every output token can only attend to earlier tokens in the sequence. So the order does matter. This is no longer permutation equivariant. Now the order actually is mattering via the masking operation.

And so now the attention matrix looks like a masked matrix. And that just means some of the arrows are not allowed. They're just set to zero. So the arrows at the top here exactly match the masking at the bottom here.

And I only allowed to learn, or I'm only allowed to use these values. Meaning, I'm only allowed to attend to things previous in the sequence. And this allows me to make predictions that are causal that only depend on information previous in the sequence. So this is what a multi-layer transformer looks like with causal masking.

On the first layer, I am trying to predict the fourth item in the sequence given the previous three. So I will not observe the fourth item, because otherwise I would be cheating. And it would be too easy.

But then after the first layer, then I can just use a mask that looks like this. Every token attends to itself and all the previous tokens. And this will achieve that the receptive field of this output token that will predict the next word can only see every previous word.

And the receptive field for this output token that I could use to predict the second-to-last word will only see everything previous to itself. And therefore I can just supervise this thing to output a sequence of words. And every word in that output sequence will only depend on the previous words. And this is an efficient way of training such a system.

I'm going a little quickly for the sake of time, but there's more details on this in the book chapter. This lecture follows the chapter almost exactly, because finally I had all the material ready for this year.

I'm sorry to say, this is just probably the worst diagram in all of science, but it's also the best paper in all of science. So it's like they balance each other out. "Attention Is All You Need," it's a super important paper, because it's the one that defined the transformer architecture.

Again, it's not new ideas-- it's just a particular recipe that is novel in its details, but works incredibly well, and a few new ideas here and there. And so this paper, just super important-- "Attention Is All You Need."

And a lot of you will come across or will have seen this diagram. This is their diagram for what a transformer is. Super confusing to me. I don't know why they call this like a feedforward layer.

But what is it? So this is the token-wise nonlinearity. That box in blue they call feedforward. So it's like just applying independently and identically an MLP to every single token vector.

What is this thing here? This is that autoregressive part that I was describing. So that has nothing to do with really transformers. That's more about autoregressive modeling. So they factor those two ideas separately, but they put them together in the first paper.

So that's just saying, shifted right. Shifted right is this thing where I just shifted the input sequence to the left, I guess, in my diagram. And positional encoding, that's meant to be like a sinusoid positional encoding.

Then all of these things, add and norm, that's just the residual connection. So just think of it like this and I think it will be simpler. And I definitely prefer this diagram instead. But for language, I would just shift the input to the left and do causal attention.

Last slide is going to be just showing now how I can put together some language modeling and image modeling to do more complicated things. So here is a case where I will take my input, which is an image. I'll tokenize it by mapping each of these patches into token vectors. I'll do self-attention and some computer vision stuff here, but it's all generic processing of tokens now.

And then I'll do this thing called cross-attention. So now I'm going to try to be describing this image in text. The task is to take an image and turn it into text. Write a caption for the image.

So I'll take this as input. And then I will autoregressively start writing a sentence where every next step in my prediction problem will be capable of attending to the previous words in my sentence, and additionally all of the representations of the image tokens.

And cross-attention is just referring to when you are having these tokens attend to these tokens, which are not themselves. They're like a different set of tokens. So you can have all different types of attention-masking strategies. And this one is popular for image-to-text captioning.

We'll come back to generative models and autoregressive models a little bit later. And I'll end there for today.