[SQUEAKING]

[RUSTLING]

[CLICKING]

SARA BEERY: So today, we're going to be doing our last of this mini series on machine learning architectures or deep learning architectures. So previously, we talked about CNNs, GNNs, things like transformers. And today, we're going to be introducing the concepts of memory and sequence modeling.

And we have seen different types of ways to model sequences in the past. We talked about 3D CNNs that have time or transformers that have a sense of time. But today, we're going to be introducing a new class of types of models that handle memory and sequences in a different way. And this will be the first class of architectures that we talk about that's actually, quote, unquote, Turing complete. So this is actually the most general framework of the architectures that we're going to be introducing.

So we'll first motivate again why we think we might want to model sequences, model something like time or order. And then we'll talk about how we might use CNNs for sequences. We'll introduce the concept of an RNN, or Recurrent Neural Network. And then we'll go through an extension or a specific subset of RNNs called LSTMs. That stands for Long Short-Term Memory. And these actually seek to specifically avoid one of the common issues with RNNs, which is forgetting. And then we'll introduce the idea of sequence modeling and long memory and start talking about some things like autoregression.

So if I look at this-- this is a single frame from a video-- there's a lot of things that computer vision can understand from this frame alone. So where for example-- and our human brains kind do the same thing. So if you look at just this frame, can anyone tell me what this is a picture of? Anything?

AUDIENCE: Kids playing.

SARA BEERY: Kids playing. Yeah. And maybe they're playing in a classroom. So you can understand something about the scene. This looks like maybe a kindergarten classroom. We have computer vision models that can understand stuff like objects that are seen in that single frame-- television, person, chair.

And we can start doing things like question answering, talking about specific attributes of those objects. What color is the chair? That chair is red. But so far, none of the things that we've talked about have really explicitly tried to understand, for example, relationships between things and how we might predict into the future.

So for example, what will the girl do next? That's something that from this given frame-- I don't know-- probably we have some ideas about different possibilities, but we might not even know, because we also rely on temporal signal. We rely on context and information. But if we watch this video, and we get to the point of that frame, we can guess what's going to happen.

[CHUCKLING]

So I don't know if any of you have ever had this experience of a chair being pulled out from underneath you, and it's incredibly disorienting when something is not how you expect it might go. But watching this video, there hits a point in the video where we're pretty sure that the kind of likely thing, which is that the girl sits in the chair, is not going to happen because the guys moved the chair. There's these complex relationships between the actions.

And then things like predicting she's going to fall, that she believes that the chair is there, why she believes that the chair is there, whether you guys are going to laugh at it. And so all of this is really very complex and nuanced understanding about the scenes, the objects, and things like intent, which we could argue about whether these machine learning models that model sequences are really getting at understanding things like human intent. But they do start to try to get at this sense of what is going to happen next.

So sequences can be a lot of things. We talked previously when we introduced CNNs about the idea of adding a temporal dimension to things like images. So now, you have a video. This might be pictures from an Italian city square over time. You can also think about sequences of language. "An evening stroll through a city square" is a sequence of words. Or you could think about sequences of audio signal. Me saying "an evening stroll through a city square" is a sequence.

And looking back again at the architecture that we've gotten very familiar with, convolutional neural networks, we can extend those into the space-time dimension and make like a space-time cube. And again, here, we're showing this in three dimensions to be for simplicity. But actually, it's really four dimensions. We're extending this fourth temporal dimension, because we have multiple inputs of red, green, and blue for each of those dimensions.

And then time just becomes another spatial dimension in that tensor. So now, you have a 4D tensor instead of a 2D tensor. And so now, when we're thinking about how we might actually try to represent time, there's different ways that we can think about capturing time within this cube. And one interesting way that you can model temporal structure can be something like taking a slice through that space-time cube. So this is what it looks like if you build this new image that takes a single horizontal pixel slice and then runs that through time. So what are we looking at here? What are these stripes likely to be?

AUDIENCE: People walking.

SARA BEERY: Yeah, people walking, and walking at different angles to the camera or with different trajectories or different speeds. Or you could also think about taking a vertical slice through time. So now, this is showing, for a given position horizontally or vertically in the frame, who is going through that position first.

Interestingly, even from a very long time ago, they started modeling images like this for the Olympics when they were trying to figure out who was the first person to go across the finish line. If you take the finish line, and you make that a single pixel element, you can see exactly what pixel crosses first, and that represents the person who won the race.

And so there's lots of different ways you can think about representing or modeling time. And then if you actually want to think about how you do convolutions in time, if we take a very simple 1D perspective of this-- so now, imagine we're just talking about this scalar perspective. Then, what you would do is you would, similar to before, learn some weight matrix over some given fixed time window. So in this case, three elements in that sequence.

And then you can think of this as a sequence-to-sequence model. So instead of before, where we took an image or something, and we would run this across the image in these different patches, and then you were constrained by the structure of the image, here actually, now, you can run this in time. So now, you're taking input from these three different positions in the sequence, and you're generating an output position.

Now, you could slide that, and you could slide it arbitrarily long. You can take any arbitrary length sequence and you can generate an arbitrary length sequence. So these convolutions in time, they're not constrained. I mean, they're only constrained by the length of your input sequence. And you can imagine that if you had something like a static camera that was taking video of this classroom continuously, that you could just keep running these convolutions in time, and there wouldn't necessarily be a point where you needed to stop or were constrained by the architecture itself.

Now, if you consider something like more dimensions, things like these RGB videos or add even additional dimensions in that RGB channel, something like MRI that captures 3D volumes over time, or hyperspectral satellite images that can have something like 384 different bands, you can continue to think about these convolutions, but now where time is potentially this infinitely long dimension.

And if you're doing this, then what we discover both empirically-- and also it kind of makes sense intuitively-- is that as that time sequence becomes arbitrarily long, it's hard for our memories of what happened at the beginning of the sequence to persist. And one of the reasons for this with a convolutional approach is the following.

So here, say we have our temporal convolution, and it's taking in some input. We're showing the scalar version, but for the sake of intuition, this is a picture of my cat Frank, who is the cutest cat in the world. And so say I'm walking around my house taking a video.

And I start at this point in time. And then I go forward. And now, in the future, I have again-- remember, because it's convolution-- that same weight matrix. And now, I want to understand what's in here. But the thing is I don't really have, from this fixed time window, any inputs coming from the beginning. I've only got the window that I can look at.

And so, based on the information I have-- well, maybe I look at this and I predict tiger, because it's orange and it's outside, and cats are usually indoors. So the idea of something like memory or trying to extend what we forget, move beyond these fixed time windows-- is to try to capture that context, be able to remember what you saw in the past.

So here then, maybe you'd have some memory unit, and that would actually give you some context, give you some information. Since I'm walking around my house, it's unlikely I'm going to see a tiger. But I did see Frank before, so it's maybe still Frank. And so this type of identification with the context of what you've seen in the past from the same, for example, sequence is really the intuition or the motivation behind the development of what we call recurrent neural networks, so these RNNs.

So transformers, CNNs, everything so far mostly had of fixed window in time. There was transformers these days-- we can talk about that context window is getting very, very large, but it's still fixed. But a recurrent neural network makes that infinite. And the way that it does it is you introduce the idea of a hidden unit. So now, you have-- for example, from this convolution you send-- you run your model through the convolution. You get some value.

And then you learn a way to take that value, that hidden unit, and predict the output. But you also take that hidden unit, and you send it forward in time to the next time step. So here, now, as you're convolving through time, you're actually taking in both that new temporal input and your memory from the past.

So here, if we want to look at this in a little bit more detail, again, just thinking of everything as scalars just because it's really easy to visualize this way. So a version of a scalar temporal prediction problem might be something like predicting temperature in Boston tomorrow. So here, now, what you're doing is you're representing time as some of discrete sequence, though you can apply these to continual sequences.

And the next hidden state-- so now, we have our inputs running through some model to get some hidden state. And then we send this output, and we also send that hidden state one time step forward. So here, it's as you're kind of running this through, what we do is we say your next hidden state is represented by some function f that will depend on the previous hidden state and the current input.

And then that f, that function, that's shared over time. So that function is, again, fixed over time. The way that we push information forward is fixed over time, and that's something that gets learned. And it's independent of time what that function is. And then similarly, we'll have learned some function g that will map from our hidden state to our output prediction for any given time step. And that's also going to be shared over time.

And so here, you'll note that this is the first time we're not seeing a DAG. This is not a Directed Acyclic Graph. We have a loop. We have a cycle. Because you're going from that hidden state to the hidden state again. And that's recurrent. And so you can write this recurrent function for what your hidden state will be, that some function of your previous hidden state and your current input beta. And then again your output function is going to be g mapped from your hidden state.

I think it's a little easier to interpret when you think about the computation being unrolled, because then, actually, for any given time point, it's still a directed graph. It's just that you have this recurrence loop. So again, this is the simplest version of this. So now, you assume all of those functions are linear layers, which we're very familiar with at this point.

So now, what you're learning is you're learning u, which is mapping your input to the next point. You're learning some v that's mapping your previous hidden state in as well, and you combine them. And then you're learning v that maps your hidden state to the output. And so now, here, you have something like this right. You'll have some non-linearities in there in the simplest form.

So your new hidden state is just non-linearity applied on top of w times the previous hidden state, plus u times your input data at the current time point, plus some bias term. And then similarly, the output again is some other nonlinearity times V plus the current hidden state plus a different bias term.

So if I got rid of W here, what would this be called-- if I got rid of the kind of recurrence dimension, if we were just looking at that mapping between the input and the output? I mean, since it's scalars, it's a little silly. I mean, it's like a one-dimensional convolution. But imagine if you extend these beyond scalars. It starts looking a little bit like that multi-layer perceptron that we're very familiar with. So the only difference here is that recurrence that's actually sending this information forward over time.

And really, I think, the point here is that the hidden state is intended to capture the most relevant historical information. And what we're learning in W is how do we actually understand what is relevant? What do we need to keep? And of course, this can be extended to be deep. You don't need to only have a single hidden state. You can actually stack a bunch of these on top of each other-- have many, many different hidden states, hidden layers. And then, now, these could be any dimensional tensor. And there are many, many, many works that have looked at different ways to structure these deep recurrent neural networks. But how do we do backprop if we don't have a DAG? Oh, yeah?

AUDIENCE: So the recurrence step-- how many time steps do you have? So you're unrolling it, but how do you know when to stop?

SARA BEERY: To infinity and beyond.

AUDIENCE: Yeah. Is it a hyperparameter? Is it something that gets learned in the network to see what's the optimal number of these--

SARA BEERY: So the way this is structured, it doesn't have a sense of stop. I mean, it'll just stop when you stop doing it. Like, when you stop putting in input data, I guess that's when it stops. But there isn't a sense in this model necessarily, as it's structured, that has an end. Because you have that weight that's passing the previous hidden unit into the next one. So if you don't give it a next step, I guess that's the end. But it doesn't know what's stopping it.

AUDIENCE: In practice, are you-- is your input like a sequence, or are you feeding it in one-by-one?

SARA BEERY: I mean, it depends on the problem. I mean, if you're using these types of RNNs to do something like model video or model audio clips, they probably have an end. So you just stop running them when you get to the end. But you could also think of it in a continual way, like maybe you're continuing to get temperature data every day, and every day you predict the next temperature, and that's going to go until the world ends. Does that make sense? Yeah?

AUDIENCE: How does this architecture capture the infinite life of the memory? Because in a way, it seems like a Markov chain. It only depends on the previous state.

SARA BEERY: So the question was, how does this architecture capture memory throughout that very long infinite length, because that hidden state only depends on the previous state? So it's recurrent. So this state only depends on the previous state. But the previous state depended on the state before that. And the previous state depended on the state before that.

So you are capturing in every single hidden unit as far back as you have. But we'll talk about how even though that's theoretically true, that this construction actually makes it really difficult to learn things across very, very long time horizons, mostly because you have some limited capacity there. And then we'll also talk about actually some of the mathematical limitations of this formulation. But yes, great question.

Cool. So how do you do backprop if we have a DAG-- if we don't have a DAG? So the simple, practical thing-- historically, this was something that really tripped people up. Backprop is intended for feedforward networks, and now we have loops. But the hack is you just decide when you're training what your fixed time window that you're going to propagate back through is. And now, you unroll it through that time window.

So you'd say, guess what? My time window is going to be this far. I'm going to go from V back to x0. And then, now, you've unrolled it. And of course, you have some shared parameters in that network, but it's actually now, again, a directed acyclic graph, because we're stopping the point where you're going to keep going back forever.

And so here, you can actually just-- again, if you're going to try to calculate the influence on that output, d of xout with respect to d of xin over from 0 to time t, you can do that by just sending now through this directed acyclic graph back to that zeroth element just by unrolling as far as you want to unroll. And that's a parameter that you would select or tune. How far are you going to send your backpropagation?

And then once you've unrolled it, and you've kind of decided on this fixed time window during training, then you can just use the same chain rule. And you can calculate what that gradient is going to be. And the interesting thing here is it does limit maybe how well we might learn things further back in time because, of course, the information is still available. At inference time, you're still going to have that previous information getting passed through or getting encoded. But if you only trained the model to send signal back to a given fixed time window, you could see how intuitively it might teach you to only capture some of limited temporal window of information. Now, in practice, there's also other complexities as we increase that time window in terms of what that means. Yeah?

AUDIENCE: [INAUDIBLE] increase the time window, is it then harder to predict if you only use a couple of inputs at inference time?

SARA BEERY: Oh, interesting. So the question, which I think is an interesting one, is if you've trained this model using backprop over maybe a time window of 20, now if you only have a sequence of three inputs, is that actually limiting your ability to do it shorter? So potentially no, because this is just looking at one component of that gradient.

But actually, if we think about this, you-- so this gradient is also going to be-- this is just the input of that on that, and that's like the end of that time limit. But in practice, you would also do everything up to this time point. And we'll go through that in the next couple slides. So you would still be getting that signal from everything within the time window that you're looking at, not just going back to the farthest point. Which makes sense, because it might be much more helpful to have some of those more frequent, more recent gradients than maybe the further ones. Yeah?

AUDIENCE: To go along that point, for this example, would it be like if xin is today's temperature, xout is six days in advance temperature. But we are actually inherently predicting one day, two day, three day, four day, five day to get there.

SARA BEERY: Yeah, so what we're showing here is just the fact that if you do unroll this, that now you do have this directed graph again. But then like in practice, essentially, what you're saying is you would only take the gradients up to t minus whatever x0 was. So this is like 1, 2, 3, 4, 5, 6.

So you'd only be taking those gradients over a sequence of six. And then, now, for the next point you'd shift it right. So now, you'd only go to x1. And then for the next point, you'd only send gradients from x2. So you have some fixed time window that you're unrolling that you actually accept gradient-- you calculate gradients over. Yeah?

AUDIENCE: The idea that the truncated time window is some approximation of an infinite time window?

SARA BEERY: I think that's a reasonable assumption. So the question was is that fixed time window a reasonable approximation of an infinite time window? I would say it's maybe a bit of both. So again, I'll walk through it in a sec, but basically, with this formulation, the longer time window you have, the harder it is for information to actually make it that far with these weights. So at some point, actually, extending the time window is probably, by construction here, not super beneficial. Yes?

AUDIENCE: I wonder, is there a way to optimize what's the time window you need for a specific task? Or is it just--

SARA BEERY: Well, it's a hyperparameter. So when you look at the literature for this, people would ablate that hyperparameter, for example. And I think sometimes, it might be something that you could actually bring some domain knowledge to decide what actually makes sense here.

So a few conceptual things. If you are going to unroll this, remember that function that maps the hidden states forward through time is fixed over time. So those W's are shared. So now, what does that mean for backprop? So we have this sequence of videos. Now, the loss is going to end up being the sum of the losses from each of those predicted frames over some sequence.

And to optimize, you want to learn how you would change W, U, and V so that the sum of the kind of loss over that sequence is minimized. This is how we do gradient descent. But you want it for the entire sequence within, I guess, the time window that you're considering.

So we want to figure out how we want to change all of these. But of course, W is, again, shared over time. So we have to do backprop over shared parameters. So we're going to sum up all of these individual losses. And now, we're going to be passing this gradient back.

And now, does anyone remember what it looks like when we have these backprop over shared parameters? Because here, we have the loss is all getting summed up, the gradients all coming down. And then there's this kind of waterfall backwards where those gradients are passing through all the W's. So does anyone remember what we do when we're taking gradients over shared parameters?

It's been a couple of weeks. Does this look familiar to anybody? Yeah, so here, there's this idea that if you had shared parameters, and they're being shared throughout a network, that it's actually the same as branching structure in a directed acyclic graph. And so when you're doing a gradient-- when you're taking the gradient over this branching structure, you're actually just summing the gradients for the parameter each time that parameter occurs.

So for that W, we're going to just be summing those gradients over every single element that we have that W in. Cool. So here, we have W in all of these places, and now we have this branching structure. And then when we're calculating the change in that loss with respect to W, it's the sum over the change in the loss, the gradients from each of those.

Cool. And the nice thing is, if you're using something like PyTorch, this will happen automatically in autograd as long as you don't do deep copies, as long as you're pointing to the actual by reference that actual variable, then autograd handles this kind of summing over the gradients within the thing reasonably well. So make sure you're not doing deep copies, and it all works out pretty nicely.

And then I think another point is you're summing over the past inputs. So if you're calculating the loss with respect to this one, you're summing over all of those. If you're calculating with respect to this one, it's only over the ones previous to it, et cetera, et cetera, because that's only going in one direction. Any other questions about backpropagation through an RNN? Cool, let's move on.

So problem of long-range dependencies. We talked about, well, as this time window starts to stretch towards infinity, it seems like it would be hard for this thing to remember. Why not set your time window to infinity? Why not remember everything? So it turns out, first, memory size will grow with that t. So if you need to use some of infinitely long sequence as an input, that means it takes up a lot more memory.

And that time and memory is non-parametric. There's not like a finite set of parameters we can use to model infinite time. RNNs make like a Markov assumption, which is to say that the future hidden state only depends on the immediate preceding hidden state. So we say, it's recurrent, but we only depend on just the immediate hidden state before that. And then by putting the right information into that hidden state, you can model dependencies that are arbitrarily far apart in theory, because the model learns with W how to keep the relevant information, or at least that's the hope.

So here, what you're saying is, actually, in order to capture those long range dependencies, we have to propagate that information through a very long chain of dependencies. And this ends up being a bit difficult. So let's walk through this-- this very big chalk. So for simplicity, just to walk through what the math looks like, we're going to assume we don't have any non-linearities, and all the bias terms are set to 0.

So say we have h sub 1. And that's going to be equal to, like we said before, W of H sub 0 plus U of X sub 1. So H sub 1. The hidden state at that first step is going to be equal to the weights times the hidden state of the previous step plus these weights times the new input.

Now, if we write out what this recurrence looks like-- now, if we say H sub 2, now that's going to be equal to W of W of H sub 0 plus U of X sub 1 plus U of X sub 2. So what do you notice here? Now, we have a W squared term essentially. And so now, if you do it for another time step, you're going to have W cubed, et cetera, all the way up to W to the N.

So if you have this quadratic relationship with your weights matrix, what happens for values in that weight matrix that are small? They'll go to 0. And anything in that weight matrix that's big will be infinite. So what happens is in order for this to be stable at all during training, all the values kind of need to be close to 1, which means it's kind of hard to capture really variable information in there. And then additionally, no matter what, even if you're just a little bit below 1, when you go to infinity, it will go to 0 because you have this sort of exponential relationship.

And so what this means is, often, that old observations in a recurrent neural network are forgotten, and that the stochastic gradients become really high variance. And you can either have these vanishing gradients or exploding gradients. And so this is why we started to look towards LSTMs. Yeah?

AUDIENCE: So a couple lectures ago, we talked about the spectral norm and using that to normalize weights and activations. So I'm wondering, would that fix this problem? And if not, why not?

SARA BEERY: So the question was, we previously talked about the spectral norm And using that as a way to try to mitigate things like these vanishing and exploding gradients. So you could try to build something in here that would try to normalize these weights so that they would be, again, close to 1, essentially, or not go to infinity or not. But this function is-- like, this part here, the fact that because of the recurrence relationship, you're taking an exponential in the weights matrix.

You could take a spectral norm over that exponential, but you're still going to take the exponential first. So that might help you with the exploding gradients problem, but it's not going to change the fact that the small things are going to go to 0. Does that make sense? Yeah, cool.

And actually, if you-- I thought this was kind of a fun blog post that specifically takes-- it's from a control theory professor, but they're looking at stability analysis in recursion. And so if anyone's interested in the different ways that people think about what makes things stable or not stable in a system if you have recursion, I thought this was kind of a nice blog post. Yes?

AUDIENCE: So you said gradients can explode or vanish. Do they both correspond to old memories being forgotten? And also as a follow-up, isn't it not a terrible thing that old memories are forgotten, because that's what humans do?

SARA BEERY: So the question is, if you have exploding and vanishing, do they both correspond to memories of being forgotten? And no, the exploding ones correspond to unstable training, which we also don't really want. And the vanishing ones would be old memories being forgotten. And then do we actually want to remember things over long time horizons? And that's kind of the universal question-- do we need this? And I would say that there are definitely cases where we don't.

And then there maybe are some cases where we do. So an example might be-- think about temperature. So we have these daily cycles of temperature. So we'd like to remember that the temperature got warmer in the middle of the day than at night yesterday. So if we're looking at predicting temperature on some of fine-grained horizon, you'd want to be able to remember at least periodically a day back.

But then also you have these yearly cycles of temperature, where it's hotter in the summer and cooler in the winter. So maybe if you're trying to model temperature, you also want to have this yearly dependency. And then you have things like these weather cycles, like El Niño, that happen every seven years. And so depending on what you're trying to model, there may actually be a lot of value in the longer and longer term memory if there's signal in more and more time scale. Does that make sense?

Cool. So this is why we introduced LSTMs, which is to try to deal with forgetting, to try to deal with this vanishing gradient problem. And the reason the vanishing gradient problem is more of a problem than the unstable exploding gradients is exactly like what this guy over here mentioned. There are things we can do to try to force-- to take normalization over the gradients to keep them from exploding-- different ways that we can constrain it, or regularize it, or clip it. But we can't do that when things go to 0.

So LSTMs are essentially designed so that the default behavior is to not forget. And then you have to learn to forget. And so the default is the identity. You're biased to not forget anything. And then you learn what to forget, kind of like garbage collection-- what to throw away. So that you can add new stuff.

And so if you look at this-- so this is kind of like the standard recurrent neural network. We have some non-linearity. This is kind of a simplified version where it looks like we're sending in the inputs. There's a function here that's mapping this in. And then we're sending out some hidden state, but then there'd be some function mapping that to the output. We're just only looking at the hidden layer component of this model.

And so what an LSTM does is it builds a little memory controller that decides what to save and what to delete. And it turns out this is still Turing complete. And it's actually very close to a Turing machine. It's kind of like that idea of a hidden tape that you can read to or write from. And so now, every single one of these functions that maps us from our inputs to what the next hidden state should be has this little controller built in.

And so let's go through the different components of this thing. So first, we have this cell state C. This is a new thing that we're introducing for an LSTM. And this is kind of the tape in the Turing machine. And this cell state is what gets modulated by the part that decides what to forget and what to add. So then we'll have some function, f of t.

And this function you'll operate over, applying your previous weights to the previous hidden state, and your current state, and then your biases. And that is going to be non-linearized by something like a sigmoid. So if it's large, it goes to 1. And if it's small, it goes to 0.

And so now, if you're taking this thing-- now, if you multiply by 0-- so if this value ends up being small coming out of the function, then it's telling you forget this thing. So what you want to do is give high values to things you want to remember and low values to things you want to forget.

So that's like what this component of the LCM is in charge of, deciding what to remember and what to forget. And then each element of that cell state, essentially because of the way that we're using a sigmoid and not something like a ReLU, which would be unbounded, is-- it's essentially the identity function with some masking of what we want to forget.

So then there's this next component. And this one is determining which indices we want to write to and what to write. So now, we have essentially this i that is used to determine basically which indices we're going to write to. And then the C of t is going to be how that information kind of gets added to the cell state.

And then, finally, we need to do the actual forgetting. And so this is where this comes in. What's quite different is, for the first time now, we have these multiplicative relationships right. So you're multiplying. You're taking the previous cell state, and you're multiplying it by this f sub t, which is going to be kind of either ones or 0's, or very close to.

And then so now that's kind of updating the cell state in the first place. And then you're going to add this new information. So you now have this i sub t multiplied by the new cell state C sub t. So in this case, it's removing information and adding information. So now, after you've actually updated that information in the cell state, now we're going to decide what gets output to the next hidden state.

It's not mathematically similar, but we just learned about transformers right where we have the keys, queries, and values. So in that case, when we talk about this value projection, these value matrix, that's kind of telling you how you should output that next state or how you should output the output of that thing. And this feels to me, on vibes, kind of like values. You've updated the cell states' information, and now you want to say, how do we take that and update it to the next hidden state?

And then note-- so we have now-- we have both these hidden states, which are getting passed through. But we also have that ticker tape, that cell state, that's also getting passed through. So there's kind of two things going on. Cool. So really, this important point-- the important thing that this has beyond that vanilla RNN is the fact that you have almost this gating functionality that's provided by multiplicative updates to the cell state, where now we explicitly learn what to remember and what to forget.

And then you multiply by 1, so you have this identity instead of multiplying by yourself, which removes the vanishing and exploding gradients. We no longer have that exponential. Instead, you're only ever going to do something like the identity or remove it and add something new. Because the default is the identity, it's kind of like a skipped connection or a residual connection in a ResNet.

You have this component where you could just learn, throughout this entire thing, to only ever send the identity-- just send the same information forward all the time. Awesome. Any other questions about LSTMs? This part is a little fiddly, so definitely feel free to go back through or visit this really nice blog post that we pulled some of these visualizations from, from Chris Olah. Yeah?

AUDIENCE: It's, element-wise, multiplication for the hidden state. Is that correct?

SARA BEERY: So the multiplication is basically like you're learning this gating function f that's telling you what to forget. And then you're learning this i that's telling you like what new information to add back in. And so those two things are going to be multiplied in. And they're going to be multiplied in element-wise in your C depending on what the size of C is. Yeah?

AUDIENCE: Sorry. Sorry if you've already said this. So it's going to be like an MLP or something?

SARA BEERY: Huh?

AUDIENCE: [INAUDIBLE] MLP or--

SARA BEERY: Oh, the sigmas are sigmoid functions-- literally sigmoid functions. Yeah, and the reason that people often use-- you could use another non-linearity, but the reason you use a sigmoid function here is because it's bounded between 0 and 1. So you want large values to map to 1 and small values to map to 0.

AUDIENCE: Oh, but the weight going to [INAUDIBLE] not on the diagonal.

SARA BEERY: Yes.

AUDIENCE: Makes sense.

SARA BEERY: The question was, if the weights are on the diagonal, and the answer is no. Cool. Let's talk about sequence models. So we just talked about a bunch of different ways that you can think about trying to model sequences. But there's this new kid in town of what we call an autoregressive model.

So previously, we said, we're going to have some fixed time window. We're going to be sending this memory forward. Things could be infinitely long. But the idea of an autoregressive model is quite simple, and we've introduced them before. So the idea is you have some beginning of a sequence, and then you ask it to predict what comes next.

And then this is the autoregressive part. You add that in. And now, again, you ask to predict what comes next. And this idea of continually predicting what's next, predicting what's next, this is how we think about doing modern generative AI. A lot of these things like ChatGPT, these language models will explicitly be trained to be autoregressive. So of course, there's no reason why you can't take the same framework and instead tell it to fill in the blanks or something.

So when you're actually training one of these autoregressive models, you might have a bunch of different examples of sentences, and then the beginnings of sentences and then the next word. And so then the learner has to learn from the beginning of a sentence what is the next word. And then when you're actually sampling, you would do something like send in some new beginning of a sentence, and then use your predictor until you want to stop or until the predictor decides to stop to predict the next thing. Yeah?

AUDIENCE: You mean you try-- predict something at time t? I take that prediction and pass that through [INAUDIBLE]?

SARA BEERY: So now, you predict something at time t And then you take that prediction, and you add it to now what's your input. So now, your input is everything up to t minus 1 and your prediction for t. And then you ask it to predict t plus 1, and then you add it in.

AUDIENCE: And this is what you mean by sampling? Is that why you used the word there?

SARA BEERY: Yeah, so then actually, the sampling part-- this is that's exactly what would happen. You would send in as your input colorless, green, ideas, sleep. Ask your predictor what comes next-- says furiously. And then you would take that furiously, and you could add it into. And you could say, well, what if I said colorless, green, ideas, sleep, furiously-- then what comes next? So this is this idea of autoregression, that you're using your predictions, adding them to the input, and then predicting again, and again, and again.

AUDIENCE: Is that [INAUDIBLE] Is that one way you could evaluate an RNN-- just print out these samples and see what it produces?

SARA BEERY: Yeah, so I mean, you could think of using an RNN to predict the next word. So your output would be what we think the next word might be. But what an RNN doesn't have is this explicit loop back where now that output from up to t plus 1 becomes the part of the input at the next time step. Does that make sense? So it's not innately something that you would do in an RNN, but you could totally loop it around that way. And kind of construct it. Yeah?

AUDIENCE: So with this model, the predictor outputs sort of a distribution, and then you sample one element from this distribution, I guess. But what if you--

[CHUCKLES]

SARA BEERY: Anyway, continue.

AUDIENCE: What if, instead of sampling one and adding it to the input, what if you put the distribution in the input?

SARA BEERY: Yeah, you're just begging the question. We're about to get into the probability models. But the question was basically what we said-- what I just said was, you take the prediction, and maybe that would be a distribution, and you're picking the maximum, and then you put that maximum in. What if instead you put the distribution of possible next options in? And we'll talk about different ways that people think about doing that.

So essentially, what we're talking about is-- or one way that you can think about doing this is trying to learn some probability of the next word. And you can take that and factorize the probability of that joint distribution. Basically, you're just reformulating the conditional distribution-- the probability of the next word given everything we've seen before.

This is true for any probability distribution. You factorize them out, and you get something that's multiplicative and is somewhat still autoregressive, because you're taking the probability of every previous thing up until that point. So you can think of this-- the sequence models in transformers when we talked about it in the transformer lecture last week-- as modeling for example, the probability of a sentence. And then that could be generative.

So here what this looks like in practice. So say you're trying to look at the probability of "once upon a time." That's the same as the probability of "once--" the probability of starting with the first word "once--" times the probability of getting "upon," given that your first word was "once," times the probability that you'd get "a" given that you already had "once upon," and then, et cetera, et cetera. So this is how you would model the probability of given sentence-- the likelihood of a sentence.

So what's a function that we could use that would map the probability of the next word given the previous word? Maybe some of the things that we've talked about in class-- specifically the probability. Yeah, go ahead?

AUDIENCE: [INAUDIBLE] regression or some deep thing going into logistic regression?

SARA BEERY: So you said logistic regression.

AUDIENCE: Softmax.

SARA BEERY: A softmax-- oh, yeah. So softmax takes an input and then gives us something that might not be a probability distribution, but at least sums to 1. So what is one of the ways we frequently use the softmax? When we're doing--

AUDIENCE: Classification.

SARA BEERY: Classification. Yes, exactly. So we could just treat this as a next-word classifier. And we I think we talked in the hacker lecture about how a reasonable starting point for a lot of things is just being like, let's see if we can formulate it as a multi-class classifier.

So here, then the input is "once upon a." And then you learn a next word classifier that would probably, by taking a softmax, give us some distribution over the likelihood of the next words. And so now, here, a standard multi-class classifier will use some neural net. That softmax will squish the outputs into a probability mass function. Non-negative vector sums to 1. And then you learn that by penalizing it via cross-entropy.

So how would you map words to classes? How many classes would there be? Lots. So if you use words and u-- every possible word in the English language, which there are many more words in some other languages on Earth, and you represent all those words as 1-hot-vectors of size k, k would be about 100,000.

And we know that multi-class classification starts to be a lot less easy to learn when the number of classes starts to scale quite large. So it is possible, for example, like we run some 100,000 classifiers for species in iNaturalist. There's about 450,000 species currently in iNaturalist. We train classifiers for all of them, but you need a lot of data. And it can be quite unstable.

So instead, another thing that you could do is you could represent every character as a class. So now, in the English language, the alphabet is 26. So now, we only have k equals 26. And that's a really reasonable set of classes. But the thing is, now your classification is fewer way, but the sequence prediction is a lot harder. You have to take a lot more time steps. There's a lot more predictions that you have to take.

And it's pretty easy for these things to devolve over time. Imagine it starts spelling something weird, and then where are you going to go? So ideally, you'd want something somewhat in between. And it turns out, in practice, that what's kind of a sweet spot is this idea of byte pairs, which is 2 character pairs.

And so now, you represent words. It's about 1,000-- 26 times 26, maybe plus a few more-- of these byte pairs. These tokens of these two character pairs is quite frequent, even in the largest scale language models we have. Cool. So let's talk about then what this might look like. So you have this kind of probability, this classifier for the next step, basically. And then you send your output prediction forward.

So one example of this-- this is an example that's taking in some caffeine molecule. And then you have a model that's trying to output a description of that molecule. So maybe that would look something like stimulant-- "a mild stimulant that enhances cognitive ability." So you use this caffeine molecule. You have some representation that you use to start your word generator somewhere, and then you would describe this. Maybe this we would call a molecule to text.

So one way you could do this would be something like an LSTM. So now, you have your molecule. You run it through some GNN that gives you some representation of that molecule that you can use to condition on your starting point. Then, once you have the starting point, the LSTM is just predicting that next word at any given point. And you're taking the previous things into account.

So if you're actually looking to train this, now, the training data is something like "a mild stimulant that enhances cognitive ability," end character-- stop predicting. So now, for the first time, we've introduced the idea of stopping when you're done. And then your outputs would maybe be some probability distribution or likelihood over those next words given the previous words.

And then you could train it using something like a max likelihood objective, where you want to maximize the probability the model assigns to each target word. You want it to maximize the probability of the word that was actually in the sentence. So now, you have your targets. You 1-hot encode them, and then you can do something like maximizing cross-entropy, because now you have this multi-class classification. You can train it the same way we've trained things before.

Now, one thing you might actually want to do in training-- currently, we're saying you would take your prediction, and then you would use that and send it forward to the next thing as the previous word. And you're learning from that. But actually, in training, often, we use something called teacher forcing, which is basically, even if you predict the wrong thing at a given time step, you enforce the correct word goes in as the input. So you actually send this in now and condition each next word prediction on the ground-truth preceding words instead of letting it devolve and then penalizing it for every mistake it made after it made its first mistake.

And then at test time, what you would do is now you have your starting point, and then you're going to sample from the predicted distribution over those words. So for example, "a." Now, we do that autoregressive thing, where we send that in, and then we predict the next thing. "A strong stimulant that enhances cognitive ability." But here, you'll note that after "a," we weren't super sure what was likely next.

And so the model predicted "strong." And it turns out that "strong" was exactly wrong. We said "mild" originally. And that actually is pretty different in terms of the meaning of that sentence. And so this is something where you have this autoregressive modeling-- if you make one mistake, then often, it can take you down a path in almost this tree where you make a lot of mistakes.

And so one way that people handle that is they do something called beam search, where you essentially let the model choose top k like what it thinks are the k most likely things. And then from each of those you choose the k most likely things next. You fill out this entire tree, and then you calculate the probability of the entire path along the tree for each of them, and you pick the one that scores best overall. And so then the score is the model's confidence of the entire sentence along that tree.

And there what we might learn is that actually maybe conditioned on the input that this correct path on the tree would score higher. And this type of approach for language modeling-- lots of different deep things plus LSTM-flavored things put together to try to generate text or describe images or do VQA-- this was really popular a few years ago. This was definitely kind the approach that people took 2015-ish timeframe. Of course, I think today, what we see is often people will use transformers instead. And we can talk a bit about what that means. Yes?

AUDIENCE: So suppose I'm only interested in predicting the third next word. For example, should I do it in an autoregressive way using the beam search and so on? Or should I use a model that only outputs the third word? Because it seems to me that there is some information in the two previous words that help you.

SARA BEERY: I mean, I think that's probably a question that would depend. So the question was, what if I only care about predicting the word three ahead of where I am now. I don't care about the things in between. Should I still model all the things in between one step at a time, or should I try to learn a model that predicts three words in advance?

I think my intuition aligns with yours that there's signal in those intermediate words that you would really want to propagate forward. And so probably I would not try to just guess from "a" what's the word three in advance without some context of what the intermediate words were. So I'd probably do this maybe with the beam search type thing.

AUDIENCE: So then if I want to predict the temperature in the week, do I need for every day to say [INAUDIBLE]?

SARA BEERY: Yeah. Well, these days a lot of temperature models are transformers with really large context windows. So you're actually sharing information across all the temperatures indeterminately. And we'll talk in a second about, when you're talking about these long range dependencies, what the trade-offs are when you're talking about using something like recursion, or even recursion with these memory controllers, versus using something like attention that potentially has easier ways of sharing information over long time horizons-- in more flexible ways, I guess is probably a better way to put it.

Cool. So we talked previously about the fact that we want to recognize that this is Frank even when he's outside. And just to be clear, he only gets supervised outside time, because I'm a good naturalist, and I don't let my cat kill birds. But he gets supervised outside time on our balcony sometimes, and he loves it. And I want to recognize him out there.

So here, say we had this concept of using a memory unit. So one thing you could do instead of passing memory forward as you went through the sequence is you could just let every single time step get a memory unit. You could just keep memories from every single point in time. And so then you would, instead of needing to figure out how to send the information forward in time in this constrained way, you could just directly link your old memories.

And so there are methods that directly link old memories to future predictions. Things like temporal convolutions do this. Things like attention and transformers do this. And we talked a lot about those last lecture. And then there are also these memory networks that you guys could take a look at if you're interested that explicitly try to have these memory representations or units that you can access over time, maybe through a separate bank of memories.

So if we're actually trying to model arbitrarily long sequences, we've talked about a couple of ways to do it. So we talked about recurrence where now you have these recurrent weights that are shared over time. And you have this cycle of where you're passing forward information over some time window. And the way that you pass that information could be just vanilla recursion. It be something like an LSTM. There's a lot more complicated ways to do it.

We talked about convolution. You can model an arbitrarily long sequence of convolution. But now you have this fixed set of convolution weights. Those are shared over time, but you don't get to see anything outside of the time window of that convolution. And then the last thing we talk about is attention where, here, weights are dynamically determined as a function of the data. So you have this convolutional kernel with attention weights.

So if you look at recurrence, that looks like this. You're passing that forward in time. Convolution looks like this. So you're getting that time window, but you're not passing it forward in time. And then attention will look something like this, where now the attention weights are a function of the data. So the way that you keep the information is dependent on the inputs, which is quite flexible. And then, of course, in all of these cases, you can have some sense of, at least the bottom two, of what this time window is. And the top one, we do talk about a time window for backprop.

So this is interesting, because now, if you take these different approaches-- if you use something like self-attention, or recurrence, or convolution-- they're going to be different computational complexities based on the sequence length-- so how big your sequence is-- the dimension of the representation, and then like the kernel size of your convolution, and the size of the neighborhood, I guess, if you're going to have some of restricted self-attention, like a window that's sliding instead of just self-attention over the entire string.

And this is actually from that "Attention is All you Need" paper. And the name "Attention is All you Need" is actually really trying to say that we don't need memory. All we need is attention. And here, you can see that when you talk about the number of operations, the maximal length, the complexity, there's a lot of trade-offs here in terms of what's possible.

But of course, with something like self-attention, you have this n squared. In terms of the length of the sequence, in terms of the complexity per layer, because you have to calculate the similarity of the entire sequence to itself. So if your sequence is length n, you need to calculate this n squared computation. And you can imagine that that can get really computationally exhaustive. It takes a lot of memory to store something like that large.

So there's a bunch of methods that have been developed in recent years that try to figure out how to make even larger context window transformers given limited memory. And so these are things like efficiency from sparsification, so things like the Reformer, or Performers, or Linformers that are all different ways to try to decompose attention-- do something like a low matrix approximation-- a low rank matrix approximation to get that context window computation from big O of n squared to big O of n. Or things like the Reformer uses this quadratic dot product attention that uses local hashing to get it to n log n.

And this is definitely something that people are quite interested in. But also frequently, this comes at the expense of a little bit of performance. And so if the trade off is performance versus computational efficiency, we just build GPUs with more memory. And then there's other ways that people try and look at even larger context transformers. So instead of-- yes?

AUDIENCE: [INAUDIBLE] less performant but more efficient, can't you just use the extra time that you gain to train it [INAUDIBLE] and get a better prediction?

SARA BEERY: So the question is, if it's less performant but more efficient, could you use the time on the GPU to train on more data? I don't think that's actually the choice that they're making. I think they're going to train on all the data they have with the biggest thing they have.

[COUGHS]

Sorry, guys. So another mechanism that people have used to try to increase the context window of transformers is this idea of local and global attention. So things like Transformer XL uses segment-level recurrence and then these very fancy positional encodings to increase the context window. Longformers scales with sequence length by deconstructing local and global attention.

[COUGHS]

Sorry. This is from Longformer, I believe, this figure at the bottom. So what you can see is that they're basically sparsifying that n by n computation. So the previous one was sparsifying via decomposition. This one is sparsifying via some of masking. But essentially, ways to try to figure out how you don't need to do n times n computations.

And then another thing that was introduced to try to increase the context of transformers without needing to scale memory with n squared-- was this idea that maybe something like a large GPT that will jointly model all the world's language and all the world's information could be broken down or decomposed into something that models just language, that's much more lightweight, with some of lookup table or memory bank that is able to let us do fact lookup in a database.

And the other benefit of something like this is there's ways to, again, ensure factual validity. Because here, we're predicting what's likely, and it's learned based on the world's information. But you can't guarantee that it actually matches the truth of the world's information. Whereas if you keep the knowledge base, this kind of lookup of facts separate, then you can try to enforce that those facts are actually going to be true, that you won't have things like hallucinations. Of course, there's still some complexity there.

And so this is like retrieval enhanced transformers, essentially-- so things like this paper called retro. And this gives you this trillion token database that just gets stored in memory and gets accessed by your language model. Cool. So how far back can we go though? Is this actually a limitation? Do we need to decrease our context window for something like attention? How much memory do we need.

And it kind of gets to this question, do we really care about modeling things at the beginning of *Pride and Prejudice* by the time you get to the last chapter? Do we care? And if you look at the attention window of some of these models as they've kind of scaled, BERT had 512 token attention-- or context window. GPT-2 had 1,024. GPT-3 was 2,048. GPT-4 was, by default, 8,000, but they had a souped up version that could do 32,000.

Anthropic has a model that has about 100K, so it's like 75,000 words. That's like two or three books worth of context. So if we can fit this in memory, do we need anything else? Well, one way that we might is how many people can afford H100s or whatever the best and brightest version of a GPU is?

So from my perspective, I think figuring out when we actually need this much context, and how we should use it, and in what ways is valuable from the sense of efficiency and accessibility of these models. And this takes a lot of power, and it takes a lot of energy, and it's really slow to train. And so I think efficiency gains on in terms of how well we model time and how we think about what's needed to model in time, I think, are really valuable.

And there's a nice paper-- it's maybe two years old now from Jitendra Malik's group-- that talks about when do we actually need long-term context? And they do this within the frame of reference of video action recognition. So if you have a video, how much temporal information do you actually need to recognize the action? And what they were pointing out-- so they introduced this idea of a minimum certificate set, which is essentially the total amount of frames or little clips of a video that you need in terms of time to recognize an action accurately.

And so here, if you have this entire video, and you want to recognize what's going on in the video, basically, how many of those frames of the video are needed? And that's the amount of context you really need. And what they showed was that most of the video action recognition benchmarks we actually study needed somewhere up to maybe like two seconds of context.

We didn't need models that could handle three-hour long movies, because actually, the actions we cared about in our benchmarks were like two or three seconds. And so their point was that what we need is we need to develop benchmarks for things that actually need long term context in order to encourage the community to do work that shows benefit there. Because it's really hard for us to measure progress on something that we're just not measuring.

And so they introduced some new EgoSchema benchmark that had certificate lengths of more like 100. But I think that this is an interesting point just from a meta perspective. It kind of tells us about how the benchmarks that we build, the ways that we model value, we model improvement in the machine learning space directly influence what we make progress on, because we want to optimize these benchmarks.

And so if there's something that our benchmarks are missing-- if there are gaps-- that means that we don't know what we don't know. And so we might say, oh, well, we want to build these models that have really long context windows. But then they're just-- the more context windows we have, or the more temporal context we have, it's not really improving performance. Maybe it's because the benchmark we're measuring performance on just doesn't actually include anything that needs it.

And then finally, I want to talk a little bit about just this different perspective on fast and slow memory. So here's one interesting thing. If you think about a neural network as something that's frozen, and you send data into it, then the activations that you get through that neural network, there's some statistic of that data point. You can think of them as like fast memory. So they're actually building you some representation that's very fast, that's kind of capturing memories of how that data is running through the network.

Similarly, if you're learning the parameters of a network via training from a large data set, you can think of the theta, the actual parameter values in that network as representing some statistics of that training data. So in a way, those parameters in a deep net are slow memory of the training data. And this is interesting. So we can think of parameters are maybe slow memory. You're slowly building this memory of the entire training data set.

And activations are fast memory. It's some statistics or some memory of some existing data point. But do we ever think about fast memory weights of a model, or slow memory activations of a model? And just as something to give you food for thought, there is this idea of hypernets that were released, which are nets that output weights of other nets. They're used as a version of neural architecture search.

And so those weights are almost like a fast memory of the input data to that net. You're getting as the output of your network another network that's a set of weights. Or there are these ideas of codebooks. These actually use tensors of activations that are learned. So you're doing backprop to the activations themselves. And you could argue that these activations are kind of slow memory of the data set you're learning.

So I think there's ways to flip the perspective a little bit and try to think about how we have memory that's stored in the weights, in the activations. What are the ways that we're representing what the data we're learning or the data we're actually using at any given point in time. Cool. Any other questions? Otherwise, I'll just leave you with another picture of Frank, who's the best cat ever.

[APPLAUSE]

Awesome. I'm happy to take questions up here. Otherwise, see you guys on Thursday.