[SQUEAKING]

[RUSTLING]

[CLICKING]

SARA BEERY: So, today, we're going to start off the first of a series of lectures on machine learning architectures. And so moving beyond the very vanilla architecture we've been talking about so far, which is the multilayer perceptron, and starting to discuss different ways that we can encode structure or essentially build hypotheses about the space of models that we want to optimize over directly into the architecture of the model that we're going to optimize.

So what that's going to look like today-- first, we're going to have a broader discussion about why we want to build better architectures and what makes an architecture better. And then we're going to do a bit of a deep dive on convolutional layers as one specific mechanism you can build into an architecture, one that works particularly well for grids.

And this might be a review for some of you, but we're still going to spend some time on it. Then we're going to talk a bit about pyramids and the idea of receptive field. Then I'm going to go through a bit of a high-level overview of a bunch of different architectures and architecture components that have made up a lot of the progress in the last 10 years or so.

And then I'm going to finish with a first touch on neural fields and positional encoding. So we'll talk a lot more about this when we get into lectures on transformers later on.

So this is what we've been talking about so far, the multilayer perceptron. So this is, at a high level, the building block is this linear combination of neurons or linear combination of inputs, a neuron wise or a point-wise non-linearity and then another linear combination.

And one of the benefits of this, like got described in the last lecture, is the fact that it can-- it's a universal approximator. You can build any model and prove that you can build any model with this type of architecture.

Another benefit of a multi-layer perceptron is it's quite simple. And one of the things this simplicity gives you is the ability to write really elegant theory about this model. And this is one of the only models that we actually have really elegant theory for because the complexity is such that we can make reasonable assumptions, it maps well into our mathematical tool blocks.

And then another huge benefit of the multi-layer perceptron is it's what we call embarrassingly parallel. Essentially, this means that this type of architecture can be really easily parallelized in terms of computation, which means we can make use of things like GPUs that enable a lot of parallel multiplications.

So one obvious example of that is that that neuron-wise or that point-wise non-linearity, that's calculated independently for every single output neuron of that first layer. And so that can happen completely in parallel.

There's no dependence across the data points. Another nice example is that linear combination of neurons, that densely connected layer or that weight, that can happen in parallel across a very large, in some cases, batch of data. That's done independently per data point.

So what are some of the bad things about the multilayer perceptron? So first one is that it has very weak inductive biases. And so what we mean by this is that the model itself, it's a universal approximator. But there's not a lot of structure or intuition baked into this model. It doesn't assume much about the structure of the model you want to learn coming out.

And so one of the things that implies is that though you can use this model-- so you can use a multilayer perceptron to approximate any of mathematical function. It might be very difficult to do so given the data that you have. It's very sample inefficient or data hungry.

This basically means that in order to get this multilayer perceptron to learn maybe a very complex function, you might require so much data that it's not actually tractable, like more data than exists on Earth to be able to really generalize and robustly learn a function with this particular architecture that generalizes well.

And then another downside is that dense layers, those fully connected layers, particularly if you want a model that's very wide because maybe the function you need to approximate is quite complex, those can be really expensive.

If your input data is an image, for example, a high resolution image that you might take on your iPhone today, then you need to flatten that out into one long vector. That's going to be thousands of input dimensions and 1,000 by 1,000 multiplication in a dense way is a very expensive computational cost.

So why use other architectures? Well, one, we might want to address some of those downsides that we talked about. But let's take a meta view of this and zoom out a little. So if this is the set of all possible mappings from X to Y-- so this is like all functions is contained in this little gray box. And maybe this is the true solution we're trying to learn. This is the correct model for our problem.

Then possibly given any data set that we're going to learn from, that data set will constrain the space a bit. So not every model will fit the data that we're using to learn the data. And so now maybe you have this subset of all mappings that actually fits your training data.

And then within that, based on something about your optimization scheme, you learn some solution, and it will be in that space. It will fit the data, but it won't necessarily be very close to the true solution.

So what would happen to this high-level diagram, if we add more data? Anyone? What would happen? Mm-hmm?

AUDIENCE: [INAUDIBLE]

SARA BEERY: Yeah, exactly. So that green space, that'll get smaller. There will be fewer models that fit all of the data if we have more data. So that's kind of nice. So if you can add more data, then potentially you shrink the space you're optimizing over, and then you'll learn a maybe more optimal solution or something that's closer to the true solution because the space is different.

But what if you don't have more data? That's not an option for you. So then another alternative is to define a hypothesis space. And this is adding some structure into the model itself. And that hypothesis space is essentially going to shine a spotlight on the space of all possible models is the way we've kind of denoted it here.

But essentially, it's a stronger inductive bias or a stronger prior over what we think the model is going to take the form of. And then if you have that intersection of the models that fit the data and your inductive hypothesis, then you might actually be able to use less data, but still find a better architecture.

So the idea is if we can pin down or if we can get closer to the true model by either adding more data or by using a more constrained architecture-- and you can imagine that the combination of the two would be even better. So more data helps. Better architecture or stronger inductive bias helps. And those two things combined is ideal.

So let's look at this from a very, very simplistic model. This is a one-dimensional model. It's a mapping of f of x, where x is a scalar. So here if we use, for example-- these are true outputs. If we use a 5-layer ReLU-net, so essentially densely connected layers with point-wise ReLU-based activations.

And we fit that to a single data point. Well, now the space of all possible models that fit that data point is very, very large. And the one that actually gets learned is just this flat line. So that's not necessarily a very good learned solution.

As we start adding more data, you can see that you maybe start to fit slightly better in the region where we add data in the middle, but it's still not a very good solution. And then as you start to add a lot of data, what you see on the right is you start getting really, really nice fits in the distribution where you have data, but you get really poor generalization out of distribution.

And so the intuition here is really that model, hypothesis, that hypothesis space you build, that's really something that helps us generalize out of the distribution of the data we have.

And so what if instead we had a really strong prior over what the model was supposed to be? So in this case, y equals ax plus sine bx squared. That's very specific. It turns out that it's exactly the correct model for this function. And you can see that if you only have a single data point, that really good model is insufficient. You still aren't able to figure out where in that set of hypotheses the correct model is.

But even with very, very, very few data points, we start getting a really nice fit. And, actually, even when we add a lot more data points, we get really strong generalization out of distribution. But the difference between those two in the middle is actually not that big. So we can get a really nice fit even with just a little bit of data when we have the exact precise correct hypothesis.

And so that's what we're showing. Architectures enable us to generalize outside the training distribution. And a good architecture is going to be one that can represent the true function of the data and then is otherwise minimal. We want it to be easy to search over via gradient-based learning. We want it to be easy to parallelize, fast on GPU, et cetera, et cetera.

And so here's a bit of a preview. But it turns out that better architectures can approximate important function classes more efficiently. So this is a result from a paper by another MIT professor, Vincent Sitzmann, where they introduced this model over here on the far right called SIREN.

And SIREN actually uses a sinusoidal activation space, and that fits images more efficiently. If anyone's taken signal processing, this idea might be a bit familiar. A Fourier basis is a good basis for representing images, and the fundamental building blocks of Fourier basis are sinusoids.

So if we take a look at what this is, if you look at ReLU-net or a TanH-net or these different models that are being compared here. And we're going to look at how they actually try to fit this specific image over time by sampling pixels from that image and then fitting that as the training data.

You can see that you start getting a pretty good approximation from SIREN much, much faster than you do from some of these other models. And, of course, after sampling enough points, eventually, all the models are getting reasonably good at estimating. But that SIREN model, it's a better hypothesis to start with for the image space. And so you get much more efficient convergence.

And so in this particular instance, now instead of looking at a 5-layer ReLU-net, we're going to look at a 5-layer sin-net where now those activations, those nonlinear activations, are sinusoids. And you can see here that with a single data point, it still doesn't fit super well.

But even just with a couple data points, we start seeing some periodicity that maybe starts to match the periodicity in the function. And even though we're not generalizing perfectly, we're still seeing some periodicity that seems to be moving more towards what the true function is. So why would you have that periodicity in the output space? It's kind of an obvious question.

AUDIENCE: [INAUDIBLE] sine.

SARA BEERY: Yes, it's a sine. Sines are periodic. Very nice. But so essentially, we've just added an inductive bias in our model architecture that says that the distribution should be periodic. And then it tries to find solutions that are periodic outside of the distribution where we have training data. Cool. Any questions about this? It's really just intended to build high-level intuition. Yeah?

AUDIENCE: So I guess I'm wondering how constrained is this to the number of pixels because the sinusoid period seems like it would be affected by that one.

SARA BEERY: So one of the things with the Fourier basis is that you are building out-- the basis assumes different periodicities of sine. And so you're reconstructing the image using different periodicities and weights of the sinusoid function. So that it handles some of the structure well. And it's one of the reasons that we can compress images really well with things like the fast Fourier transform. Yeah. Cool.

So let's now talk about convolutional neural networks. So here's an image. What's in this image?

AUDIENCE: Birds.

SARA BEERY: Birds, OK. So what else is in this image?

AUDIENCE: Sky.

SARA BEERY: Sky. So if I had a whole image classifier, if I took this image and I ran it through one of the models we've talked about so far, what would we want it to say? What's the category of this image?

AUDIENCE: Birds and sky.

SARA BEERY: Birds and sky. I mean, that would be nice. We haven't talked about multi-output classification, though. Yeah?

AUDIENCE: Nature.

SARA BEERY: Nature, yeah. I love that answer. So one of the things that's really interesting is you can really see this trade-off between generality and specificity. And so, yeah, it's like, oh, well, that's too hard. I'm just going to predict something coarser grained. And that's a reasonable fallback.

So what I was trying to get at here is actually for a lot of things we might want to understand in an image, maybe a single category isn't sufficient information. This could be flock of birds, for example. I mean, maybe that's a category.

But really, it's just that a single label is usually insufficient to represent the semantic concepts that are captured in images. And this is actually, in the grand scheme of things, a pretty simple image. I mean, think about this versus an image like of Times Square, for example, how complex that image signals semantically could be.

So if you were to take a picture, for example, from where you're sitting, I don't know, out there in the audience, think about what would you categorize that picture? Is it lecture hall? Is that students? Is it projector? Is it MIT? It's all those things.

But so when we start to think about these more complex semantic signals, and we'll get into a lot more of this throughout the course, one thing that starts to be more and more true is that there's often maybe different categories or different bits of information that are somewhat locally constrained. And so you can think about breaking down the problem into something that actually looks at location as well as semantics.

So one way that we could try to address the fact that this image contains multiple things is we could first subdivide the image into patches. And then we could try to classify each patch separately. So you could take this image, chop it up into little pieces, and then maybe you have a classifier.

And now you send this patch to the classifier and it says bird. And then you could run that classifier over the entire grid. And you would take all those patches independently. And you could then-- hey, bird. This one is sky. And you could densely populate that entire space and get out the category per little block or little patch.

And so if we think about this and we have some function that's going to crop this up and map every patch to a category, in some way, this is actually giving us an image output. It's a bit like a low-resolution image. It's got some spatial context, which is useful.

But there are a couple patches here where the objects don't necessarily fit super neatly inside them. So I don't know, maybe this one over here that's got like the tip of the bird's wing just in that one edge of the patch. It gets a little ambiguous. Do you want that to be that patch, that whole patch to be categorized as bird, if there's any bird at all? Do you want it to be-- what is the majority of the pixels in the patch?

And so maybe we want a higher resolution output map. We don't want to just know at this really coarse resolution. We actually want a finer resolution of the output. So one way you could do that is you could use smaller and smaller patches.

But it turns out that as the patches get smaller and at the far extreme, the patches are a single pixel. From a single pixel, you don't have enough context to determine what the semantic category is. It's not easy to recognize content as those patches get small.

So instead, the winning idea here with convolutional neural networks is we use large but overlapping patches. The very high-level intuition behind all of this is that you want context that's local to be-- you want to be able to look around the pixel you're predicting at when you make a prediction, but you also want to do this in a dense way.

So the way that would look is, OK, you have your patch size, and that patch size is probably a parameter in your model. You want to make sure that context window is kind of big enough that you maybe see the whole object of bird. But then when you're actually predicting, you're sliding this over as you go. And then maybe you're only predicting, for example, the object class of the center pixel but given the context around that pixel.

And so now your training data would be that patch and then the output category of that center pixel, for example. And now, if you actually run this categorizer over your entire image, you start to get something that actually looks pretty cool. That starts to look like, semantically, the boundaries of one category and another.

And this problem of categorizing every pixel instead of categorizing a whole image is called semantic segmentation. And here we're categorizing every pixel as one a set of categories. This can be extended. And maybe we'll talk a bit later about instance segmentation, which is a similar problem. But now you want to identify unique instances of a given category and capture them separately.

So now you have essentially the building blocks of a convolutional neural network. And one thing that's nice about this is because you're processing each of these patches independently and identically, that means that the actual model itself is invariant to translations of the input.

So this is an equivariant mapping essentially. So since you're processing every single patch in the same way, that means that it doesn't matter if the bird is in the top corner, the center, the bottom left, those pixels are still going to be categorized as bird.

And another way to think about this is if you took your function and you applied it to a translation of the input, it would be the same as if you translated a function of the original input.

And why would this maybe be a nice thing to have for something like semantic segmentation? Why does this make sense as an inductive bias or a hypothesis for images and for categorizing like this? Yeah?

AUDIENCE: Like the answer for each pixel kind of corresponds to the [INAUDIBLE] next slide [INAUDIBLE].

SARA BEERY: Yeah, yeah, yeah. So the answer corresponds to that local context window. Yeah, if you took an image of a bird and then you moved that portion of the image corresponding to a bird to a different part of the image or imagine you were actually taking a photo and then you shifted your camera a little bit, the object you're taking a photo of is the same.

So there actually is this type of symmetry innate to categories in the real world. And this symmetry is something we've built into our hypothesis space via the model structure.

And so, yeah, the world is somewhat translation invariant. A bird should look the same no matter where it is in an image. Of course, we're kind of glossing over pose, which is a more complicated factor here. But if you have something at the same pose and you move it somewhere else, it still maintains the same category. So cool.

One possible version of this, of this model, would be to compute a weighted sum of the pixels in each patch. And then the weights of those that sum, that's what we're going to learn. And so if you do this, then essentially that set of linear weights, W, is a convolutional kernel that then we can apply to the entire image.

And so what does this mean/ What does this kind of concept of convolution and a filter or a kernel? So a convolution is a linear, shift-invariant transformation of grid-structured input data. And essentially, in signal processing, convolution is a way that you do filtering.

So how many of you have taken, I don't know, a signal processing class or learned signal processing maybe in like an electrical engineering class? Raise hands. Cool, so a good number of you have seen this before. And here, the filter or the weights we use to take that linear weighted sum.

And then mathematically, if we want to structure that out here at the bottom, the output for a given location when applying the filter is going to be some bias, plus a linear combination of the filter weight for that coordinate times the pixel value for that coordinate in the image, and then summed over all the filter weights and all the pixels in the patch.

So really what it's doing is for any given filter-- like in this case, this filter is kind of diffused all around, and then you have something dark and something light. So this filter is going to look for that pattern. By look for, that's very anthropomorphized. But you're going to produce maximum output values when you have that same kind of dark to light transition.

And so what that actually looks like, if you run that filter over this image, is you start to get something that's highlighting areas in the image that match the filter the most, so match it most closely. So here we can see that we're getting really high values, so the white values along the boundaries between dark to light. And we're actually getting really low values along the boundaries from light to dark because that's the place that's going to be maximally opposite to that filter.

So in this particular case, what we've got is a filter that's finding us horizontal edges between dark and light values. Any questions? Yeah?

AUDIENCE: Can you explain [INAUDIBLE] variance transformation in this context?

SARA BEERY: Yeah. So, again, here if I took this image of a clownfish and I shifted it over and I ran this convolutional filter over it, the output would be this shifted over. And so you could shift the input, then apply the convolution. Or you could apply the convolution and then shift the output, and the results would be the same.

So now let's look at this from a more neural networky perspective. So this is a fully connected layer. This is what we've seen so far. And here all the outputs depend on all of the inputs.

And this can be quite computationally expensive when things get very large. Instead of a fully connected network, a convolutional network is essentially a locally connected network. And so often we can assume that the output then of this model is a local function of the input.

So here if we use the same weights for these linear combinations, this w matrix, if we use those weights the same regardless of where that initial input values are coming from, that gives us a convolutional neural network. And the way that we actually denote that is with this star operation for convolution.

If you have take signal processing, actually, this is not quite the same definition as familiarly used in signal processing for convolution. Usually, it has like a flipped value.

But here when we talk about neural networks, instead of calling this cross correlation, which might be what it is in the signal processing literature, we just call this convolution. One of those reasons is because in these networks, it's very easy to learn how to flip a sign, so it doesn't matter so much. Yeah?

AUDIENCE: Is it always a five-point star?

SARA BEERY: I mean, people do all sorts of stuff. We are going to use a five-point star in our lectures and in the notes. I think it's kind of pretty. So what does this actually look like?

So here we have this same linear combination of weights. We're running it over the local structure. And then you're taking that weighted sum. And so the outputs for each of these values are going to be corresponding to just that local input. But it's going to be the same weights that are used across the entire input vector.

And the nice thing with that thing is that you get this, again, embarrassing parallelism because now you can apply that same weight matrix across only smaller patches of the input. But you can do that in parallel.

And if you want to think about what that looks like in this broader sense, if you're taking a fully connected layer, that's the same as taking your input vector, multiplying it by a dense weight matrix, where every single one of those values matters. And then that's how you get your output. And now you could think about actually a convolutional layer is the same.

But now the weight matrix has a very specific structure. So instead of being dense, it's going to be very sparse. So in this particular instance where we're only looking at a sliding window of maybe three elements of that input vector, those three elements are going to be the only-- above and below that diagonal, those are going to be the only components that have values, and the values are going to be exactly the same along the entire diagonal of that matrix.

And there's a special name for this type of matrix. It's a specific form of a Toeplitz matrix. And so one of the things that's cool here is that it's essentially a constraint on a standard linear layer. And there are fewer parameters that can be learned because you're just learning essentially parameters for just that local structure.

So there's less of them, and they're going to be repeated. And so that means you can have a lot less overfitting. There's less open flexibility. And you're building some nice constraints into the system. You're making it equivariant. You're making it translation invariant.

And then another nice benefit of this is because this thing has this specific structure where you have zeros almost everywhere, except for just above and below the diagonal, this can generalize to different input sizes because you can take that same weight matrix and you can scale it. It's not going to change anything about the model you've learned.

And so that means that convolutional layers can be applied to arbitrarily sized inputs, which generalizes beyond the training data in a really nice way. Whereas if you took an MLP and you applied it to an image and then you tried to apply it to a larger image, you couldn't do it. This isn't something that you could apply to a larger image. You just wouldn't have weights for some of that size.

And so one way to look at this is like you could take that same filter that we applied to the clownfish, and you could apply it to any image. And you would always get out these edges, these specific things. So five kind of high-level views-- yeah?

AUDIENCE: If you go back just like before. Yeah. If you scale the matrix, does each pixel color here correspond to one value?

SARA BEERY: So we talked about the weights, so w1, w0, w minus 1. So that's like if you're taking this from a vector perspective, you want to look at the wait just before and just after. That means that you're only going to have three values.

And those three values, if you transfer this into a linear layer, those are going to be the values above, along, and below the diagonal for the whole matrix. So you just take that and slide like the ladder out along the diagonal. It'd be the same, like that.

Cool. So some different ways to think about convolution. So, one, convolutional layers are equivariant to translation. So this is a built-in hypothesis. Two, convolutional layers enable patch processing. It builds in explicit parallelism into our computation. It makes it computationally efficient.

Three, convolutional layers can be thought of as kind of image filters, things that are highlighting specific structures in the input data in some useful way that's learned. Or you can think of them as parameter sharing.

You're using the same parameters across the entire input data in a computationally efficient way. And the last point is you can think of them as a way to process variable size tensors because you have this built-in generalization across the input size. Cool.

So what happens when you stack convolutional layers? So here what we've got is two sets of convolutional layers. So, first, our input data comes in. We have a first convolutional layer. Then we have a non-linearity. And then we have a second convolutional layer.

And it turns out that if you think of it this way, actually then the entire CNN is a convolutional filter, but one that's nonlinear. So you can really think of this way. And so here now this output depends only on the local structure of that input. And that function, even though it has that non-linearity in the middle, is the same for anywhere, any of these local patches in the input image. Does this make sense? Yeah?

AUDIENCE: Why is it [INAUDIBLE]?

SARA BEERY: Yeah. So I'm just saying here in this particular instance, assume now that you have a single convolutional layer, a point-wise non-linearity just like we had for MLP, and then another convolutional layer.

So since we said that a convolutional layer is essentially a constrained version of a linear layer, you can essentially take it the same way. But the nice thing is that constrained version of the linear layer, it maintains that local constraint even as you start to stack layers.

But one thing that's very clear is as you're adding more depth, you're increasing what we call the receptive field of the output. So you're adding more layers. That means that the local patch that corresponds to, for example, xL of j is larger than the local patch of a single one of those same layers because you're taking an input from a larger size and then you're doing that again. And this is what we call a spatial pyramid, for example. Yeah?

AUDIENCE: [INAUDIBLE]

SARA BEERY: Sorry. Can you be a little louder?

AUDIENCE: [INAUDIBLE]

SARA BEERY: Oh yeah, yeah. So the only reason this is nonlinear is because there's a non-linearity in the middle. So from between that first layer-- so you have this first convolutional layer, and then see where you have those nodes in the middle. What I'm saying is here we've applied an activation, a nonlinear activation.

And then you have that second layer. Maybe that's something that in future years I'll just put that in the diagram so you can see that non-linearity, so it'd be a little clearer. But I'm just saying it exists. So now you have a convolutional filter, but it's nonlinear because you have that nonlinear activation that you've put in the middle. Yeah?

AUDIENCE: What is it called when there's [INAUDIBLE] established when you're pulling weights from previous dependents and previous layers [INAUDIBLE]?

SARA BEERY: A receptive field is one term that people will use. So if you're like increasing the receptive field of your model, you can do that in a couple of ways. You can make the original patch size, input size larger, so the kernel size of the filter size could get larger, or you can just add more depth. And that will also increase the receptive field. Yeah?

AUDIENCE: A couple slides back, you represented acts as a vector. So that's an image that's been flattened into a vector.

SARA BEERY: Yeah. So right now, we're talking about vectors because they're really clean and easy to look at. But, yeah, you can essentially say, you take an image and you flatten it out, though I think it's a little more complicated than that because you wouldn't take an image and flatten it out and then apply convolution like this because you would have this weird like stripe-wise structure. So, actually, we do take those convolutions in two dimensions. Yeah?

AUDIENCE: In the second layer, would you use the same kernel?

SARA BEERY: No. And that's why the colors are different. I mean, you could. But usually, we don't. Those weights are learned, but they are shared across the entire layer. Yeah?

AUDIENCE: So just to make sure I understand [INAUDIBLE] each weight matrix in each layer [INAUDIBLE] except for the [INAUDIBLE] layers in our network in order to represent the [INAUDIBLE].

SARA BEERY: So the question is like for each of these layers, now that we have a convolutional layer, they can be represented as that sparse matrix where you only are looking along the diagonal. We don't always necessarily actually compute it that way. For efficiency or for parallelism, often, we just really do compute it patch wise.

But your point was if you're now really restricting the search space by only allowing like a few parameters per layer, doesn't that mean that you would really struggle to represent images or learn? And the answer is yes. And so that's why often with convolutional layers, either we'll have quite a few different filters that are happening in parallel, we'll get into it later, or we add a lot more layers in a dense way.

But what we have found experimentally, and there's some nice theory around it, is that for things where this is an appropriate inductive bias, you can learn more efficiently. You get a better fit with less data. Cool.

So what if we have color? I mean, we're really taking a very image perspective here. But essentially, what we're talking about with color is multiple input channels, so not just spatial extent, but you have a tensor that also has something like depth. So if you have multi-channel inputs-- so say you do have RGB. That's what we're showing here.

What you do to do convolution over multi-channel inputs is you learn a different set of weights for each of those input channels. And then you sum over all of that to get the output value. So you sum essentially over the convolutions for each of the input channels to get the output value you're summing per channel.

So now what that looks like is you would still have a single output value for each filter. But now the filter is going to have per channel weights that are learned. And you could arguably force them to be shared. But often, we let them be separate.

What about multi-channel outputs? So if instead of having just a single output of your convolutional layer, you want to have multiple channels of outputs, here what you actually do is you learn what we call a filter bank or a set of different output convolutions.

So here instead of just learning a single linear layer, you're going to learn two in parallel, and each of them get to build their own output dimension. And so now you have a filter bank of let's say C different filters. And you get a different output activation for each of those filters from the same input data.

So then you can generalize this. And generally, in most of our convolutional layers in deep learning, we have multi-input and multi-output functions. And so here you'll have essentially your output is going to have some number of dimensions or some number of feature channels.

The input will have some number of input, dimensions or feature channels. And then you're going to have some number of filters that you convolve with that input to get this output. And the number of features that you have, you're going to have the number of output dimensions, number of features or filters.

Each of those filters is going to have some kernel size. That's K here, so some of spatial dimensions. And then for each of them, you're going to have enough channels to match your input data.

So let me show a few more examples of what this looks like. So here's, for example, a bank of two filters where you have multiple inputs and multiple outputs. So here you have two input dimensions. You've now learned this linear sum for your height by width or this kernel size, which in this case looks like 3 by 3. So each of these sets of weights, F1 and F2, those are learned separately, but they're shared across the entire image.

And so what those feature maps might actually look like, they get increasingly complex, I would say. So here, this is your input. This is using an AlexNet model, so a convolutional model. Now you take that input. After the first convolutional layer, this is the number of feature banks that are learned or output channels that are learned.

This initial input is convolved with this many different convolutional filters. And so now we're showing all of those different filters, what actually activates when you send in this image. And so you can see, I don't know, maybe some of them are capturing things like certain types of edges, certain types of structures.

And then after that second convolutional layer, these things start to get maybe a little bit more complex. They can often be quite difficult to interpret. But every layer can be thought of as a set of C feature maps or channels. And every feature map is essentially like structural representation. It's an image. It has height and width. Yeah?

AUDIENCE: How do you ensure that each feature map is uniquely different so that you have a set of ones that can combine?

SARA BEERY: So how do you make sure that all of the different feature maps are different? So that's not actually something that's innately built in, but one of the ways that people will do this is through different types of regularization in the model, so sometimes things like dropout that tries to make sure that you don't make the model too brittle or reliant on different things.

But usually, the model will learn each of them uniquely because that gives it more capacity. There's nothing in here forcing it to keep them all the same. And so why would it not learn to maximize the types of variants that it could have.

Depending on the data, and the architecture, and your optimization, you can have feature collapse sometimes where some stuff just gets ignored that you might not want to be ignored. And then you can try to build in different mechanisms to make that not happen.

All right, so we're going to do like a simple little quiz. So here's a layer. We have this input. It's three channels, so RGB by 128 by 128. Very standard image size for an input. And then we have a filter bank full of 3 by 3 filters.

So the spatial extent of those filters is 3 by 3. This is often the way we talk about it. So we're not talking about the channel dimensions. We just talk about the spatial extent of the size of the filters. And then our output is 96 channels by 128 by 128, so by height and width. So, first, how many parameters does each of those filters have/

AUDIENCE: 9.

SARA BEERY: 9. So why 9?

AUDIENCE: It's 3 by 3.

SARA BEERY: But how many input channels did we have?

AUDIENCE: [INAUDIBLE]

SARA BEERY: All good. Yeah. Every single one of these filters has 27 parameters. We put this in because it's a little confusing. We talk about the filters being 3 by 3. But the thing is when-- this is just convention. When we talk about filter size, we often only talk about the spatial extent of the filters.

And the assumption is we don't actually convolve in channel space, though, of course, you could. That's something you could totally build in. So maybe you have something like a hyperspectral input. You actually want convolution across the spectral bands.

You could build that into your model. But standard, we often use independent per input channel weights that are learned. And so then we only talk about the spatial extent. So the filter banks are 3 by 3.

Now how many filters are in our filter bank? Yeah?

AUDIENCE: 96.

SARA BEERY: 96, exactly. There are 96 outputs here, which means we get 96 filters. And then each of those filters will have those 27 parameters. So you're doing this-- for each of those 27 parameter input filters, you're running those over the entire image, and you are now taking a linear sum over all three of those channels, and you're getting a single output dimension. And so if you have 96 output channels, you need 96 different filters. Nice. Cool.

So high level, filter sizes, if you're mapping from some x of l, so this current layer where you have C sub l channels and then N by M output dimensions and you're going to map to the next layer where you have some C sub L plus 1 output dimensions and N by M spatial dimensions and your filter has spatial extent K1 by K2, then the number of parameters per filter will be K1 times K2 times C sub l.

And then the number of filters will just be C sub l plus 1, the number of output channels that you have. Yeah?

AUDIENCE: So I [INAUDIBLE] square filters when dealing with convolutions. I'm just curious where you have rectangular filters or [INAUDIBLE]?

SARA BEERY: Absolutely, normally, the filters are square. That is totally standard. They don't have to be. A convolution can be defined in a way that is not square. But I would say normally we see square convolutions. Yeah. Cool.

So you have say your input image. Now you're going to run it through a first layer. So essentially, for each of your filters, you have a red, a green, and a blue filter. Those get applied.

And then you can think about, OK, so now if you take just one of those filters, that's going to give you just a single one of those output maps in your feature maps at layer 1. A single one of those output maps is going to correspond to one of those filters over RGB. And now if you run that a second layer, second convolutional layer, now you have layer 2, your whole sets of features.

Now each of those are going to be-- instead of RGB, they're going to be across the entire set of feature maps in your first layer. But then one of the outputs after that will only correspond to a single one of those C2 filters. So you can see how the actual representation gets highly complex, highly quickly. Even just one or two layers of depth, when you have convolution, it can map really complex functions.

So another thing that we often use when we're building convolutional neural networks is this idea of pooling. So here, what we have is we have this convolutional layer. So you have your convolutional weights, you have your bias, and then you have your point-wise non-linearity.

But then one of the things that you can do is you can do what's called pooling over those output features. One of the reasons you might want to do that is because you can see that the dimensionality can get really large, really fast.

And so one thing, very, very common thing, is something we call max pooling, where here you just take some local window and you just use the maximum value. And you use that to represent your output. Or you can also think about something like mean pooling. And here you're just taking the average over some smaller local window.

So why do you want to do pooling? So one thing that's really nice about it is pooling across spatial locations gives you some stability. So for example, here, if you're taking the max over this window, you're still going to identify that there's an edge there.

There's still this white value that will get passed through, and that will be stable as the inputs move. So you'll get a large response to something like an edge filter. Regardless of the exact position of that edge, you get some more stability in the outputs.

And then sometimes we also will do pooling across channels. And one cool thing you can think about here is if you pool across feature channels where all the feature channels are looking for those edges but across different orientations of that edge filter, then you can get some new kinds of invariances. So for example, this would give you a filter that was finding edges and was invariant to the orientation of the edge. Seems nice. Cool.

So now if we actually look at a lot of the time what computation and neural network looks like, a convolutional neural network, we will usually take an input, and then we will run it through stacks of this building block. And this building block is some computation over some set of filters, a nonlinear activation, so a ReLU, and then something like pooling. And then we'll repeat that over and over and over again.

But this is a really large network that has a lot of density. And as we maybe add more feature channels as we keep going, this could get really big. So particularly, if you're trying to do something like classification, where the only thing you want to get out is a low rank representation of the input, just something like a word or a specific one hot encoding, here we actually often use something called downsampling, where we assume that we don't actually need that high-dimensional input structure at every dimension of the network.

And so here, instead of just pooling, we might also downsample. And what that looks like-- so here if you have pooling, you've now done something like your convolutional layer, your activation function, and then something like max pooling, then downsampling would look something like this.

So maybe you would only keep some subset of those actual values. And then instead of doing the pooling across all of them, you can also just do just straight downsampling.

And one way that you can actually do both pooling and downsampling together is through something called a strided operation. So this actually combines something like convolution or pooling and downsampling.

And the way that it does that is instead of going and sliding your filter by a single pixel every single time, instead, you increase what we call the stride of the filter. So now you operate your convolution. And then instead of just moving one, you might move two or you might move five.

In this particular example, here we're showing a stride that's quite big. Usually, for these models, we often have that stride be somewhat overlapping, so you don't want to lose any of the context from the original input.

So we'll have that stride be something that's overlapping. But maybe your input kernel is like 7 by 7. Your stride might be 5, something like that. And so here you're combining that pooling, that convolution, and the downsampling into a single element.

And you can also think about doing something that we call using a dilated filter. And essentially, all this is it's just a filter where we've set some of the values to 0 and baked that into our model.

And this gives you a larger receptive field, so what I was talking about before. You get more information from a larger spatial context or spatial extent with fewer parameters. There's less flexibility. There's less parameters that are learned, but you're kind of getting that larger context window. And that's kind of a trade-off that you can optimize for. Yeah?

AUDIENCE: Oh, sorry. So when you downsample, you cannot go back to the previous dimension to input dimension unless you give information across the network, right?

SARA BEERY: We'll talk about that in a little bit. So the question was if you downsample, you can't get back to the original input dimension unless you add some more information. That's not necessarily true. But you might not do it very well. So we'll talk about that in a little bit.

AUDIENCE: What are the current norms for when it's acceptable to use striding [INAUDIBLE]?

SARA BEERY: What are the current norms for when you would or would not use a stride? So I would say that this search space of possible architectures is vast. And even if we only consider the building blocks we've talked about so far, which is like linear layers and convolutional layers-- and then you can have varied input sizes, varied depth, various numbers of channels.

All of a sudden, you already have a really large search space. There's entire fields of research called neural architecture search that actually specifically tried to optimize over the space of architectures to find the best possible solution. Pretty computationally expensive to actually train those.

But I would say that one of the things that's kind of proven out in the literature over the last of maybe 10 years just experimentally is there seem to be some standards that seem to be trends that are long held. So for example, in the original AlexNet paper, the filters were quite big. They had pretty large spatial extent, and then there were less depth in the layers.

One of the things that we've seen more and more is now we actually often see filters that are reasonably small, 7 or 5 spatial extent. But maybe then they're actually stacked quite deep. So then you end up with really large receptive fields, even though the initial filters were spatially quite small.

And then with striding, again, it's just that trade off between the amount of computation and the receptive field. So there's different kind of optimality.

One of the things you can do is you can go look at a bunch of the standard architectures, and often they'll have these architecture diagrams. You've seen them before probably where it's like the layer, and then it'll have just like a layer size for each thing. And you can look at, what are the similarities across the structures of these different models?

AUDIENCE: Does the complexity of the picture matter at all, for example, the first example, you gave, which is Times Square versus the nature?

SARA BEERY: Yeah, experimentally, yes. So if you have really non-complex pictures like CIFAR, for example, these 100 by 100 really simple photos, then the network itself often needs to be much less complex. If fine-grained feature details actually matter a lot, which happens for us a lot with the species images, for example, then one of the things that actually tends to matter more than almost anything else is the input image size that you use.

So a lot of these models, the input image size, if you scale it up, then the computation scales up quite a lot with that. So we'll often downsample images to some smaller size before we run them through a network as part of the pre-processing.

But we find that increasing the scale of that input image and then maybe downscaling some of the computation is sometimes a better trade-off. If you really care about fine-grained features, you don't want to lose that detail. Yeah?

AUDIENCE: So normally, in signal processing, we're providing filters for a given purpose. [INAUDIBLE]. And in this case, the weights, that would be central to your argument.

You mentioned earlier the model will learn [INAUDIBLE]. So how would you-- or do we include inductive biases in how we train our models so that we can design our models to then do a certain purpose, no different than designing an [INAUDIBLE] for a signal? How does the model [INAUDIBLE]?

SARA BEERY: Yeah, it's a great question. So the question was in signal processing, we're often very carefully handcrafting our filters so that they are capturing a certain dimension of the signal that we want to capture, whereas here we're talking about filters that are actually being learned. And what's the trade-off? When does it make sense to try to actually craft the filters to match our intuition about the problem?

So what you're talking about is actually this massive paradigm shift that happened maybe 10, 15 years ago in the field where we went from most of the work was on what we called feature engineering in computer vision or in deep learning, where you would be building these really handcrafted structures that then you would combine together into some larger system that would recognize stuff.

And when we started to see that filters, just like these models being learned end to end were more predictive, it was a bit of like a catastrophe emotionally for many researchers. Because you were like, no, like I spent my life learning how to craft these filters in this really specific way. I don't want to just cram a bunch of data through a structure and have that be learned and then have it be better.

And it turns out that actually the trade-off often comes down to how much data you have. So if you don't have enough data to learn the filters, well, then more handcrafting, more knowledge, more inductive bias tends to be better because you actually can put more constraints on the system so that you can optimize better with less data.

But if you're in a space where you can get huge amounts of data to learn from, often, we don't know as much as we think we know about what optimality might be in terms of the filters that are learned, especially when you think about-- it can be really intuitive to think about crafting a single filter.

But how would you handcraft like the seventh convolutional layer in a 128 layer network for a single channel? It very rapidly gets to a place where the system is so complex that it's very difficult to build intuition about what it should be for us. Cool.

So receptive fields. So one of the things we talked about is if you let of input size of the patch get too small, you maybe won't be able to actually identify the things you care about.

So in this particular instance, say maybe the input patch is this entire bird. And then we're running it through convolutional network, ReLU, convolutional layer. And then we get some output. And that output for that patch is bird.

So here what we would say is the receptive field of that x2 output is the input patch there. Because receptive field essentially just says what parts of the initial input have any influence on this part of the output. Because we're doing this in a way that's translation invariant. We're doing the same computation across different patches.

Outside of that receptive field, this model doesn't have any input information. So if there was something just outside of that field of view that was really informative and it wasn't actually in here, then there's no way for the model to get information outside of that receptive field.

You can also talk about the receptive field of input layers. So you can say, OK, like in this first layer, the receptive field of that component of the output of that first layer is that part of the input.

And what that looks like-- as you pass through the network, you can think about looking at the activations, and you can see that they go from being very fine grained or very detailed in the early layers because maybe you're running them through convolutions that are much more local.

And then as you get further out or further along in AlexNet, VGG 16, ResNet 18, what you start to see are feature activations that are kind of much more diffuse. And this has to do with that receptive field because now each one of these output dimensions is actually collecting really complex information and mapping it through a bunch of different layers from this input.

So maybe you're getting output feature activations that are more semantically meaningful but less affected by subtle texture or variation in the initial input image possibly. Yeah?

AUDIENCE: Similar computation to how you do Gaussian blurring?

SARA BEERY: So the question is if it's similar in computation to how you do Gaussian blurring. I think at a high level, like the intuition is a little similar, but computationally, it's quite different. Yeah?

AUDIENCE: In that last image, are those, 1, 6, and 8, are-- the ones on the right, are those the same pixel size as the initial inputs, or are they downsampled and you just resize them to all be the same?

SARA BEERY: We've resized them all to be the same size. But it actually is very dependent on the architecture. So ResNets, the actual spatial extent, once you get to block 8 is much lower. And then there's a lot more channels versus the AlexNet one. Yeah. Cool.

So in the last little bit, I want to talk through at a pretty high level a bunch of these different high-level ideas, maybe some of these what one of you mentioned, some of these things that have tended to continue to be used over time. So we talked about one very, very classic version of a convolutional neural network is something like a classification network.

And most classification networks have something like this structure. We'll have filters, some sort of point-wise non-linearity, and then some form of downsampling. And we run through a bunch of different layers like that to get out some of output classification.

But another really common structure in using convolutions is what we call an encoder-decoder architecture. So here what you're doing is you're taking that input and you're doing convolution, non-linearity, subsampling or downsampling to get down to some very low dimensional representation, for example this z space vector.

And then in the decoder component, you're going to take that low dimensional representation, and you're going to now do convolution non-linearity upsampling. So in that upsampling component, you're actually just increasing the spatial extent of each of the layers going forward. And the training signal here is you want the decoded output to match the encoded input as close as possible.

Now this is also used. So this is used as one way to get a low-dimensional representation of an image. This type of encoder-decoder architecture, this is built into things like variational autoencoders.

Masked autoencoders are increasingly widely used, though those use attention as opposed to convolution as a way to share information over space. You can take these different components, and you can break them out.

So we talked about differential programming. If you train an encoder-decoder architecture, you could use the encoder to build some representation of input images. But you can also use the decoder to generate images. So these components can be used separately and can be valuable separately.

But another way you might want to use this type of encoder-decoder structure of architecture would be to do something like get out semantic segmentation because you want an output space that has spatial extent, doesn't necessarily need to be the exact same resolution as the input, though it can be. But you want something that has a spatial structure in the output space as opposed to just like a single category.

Now this seems nice. But it turns out that this is both a blessing and a curse, this bottleneck in the middle. It's forcing the model to learn something that's kind of semantically useful, but it also means that often it's not possible with an architecture that just has downsampling, then upsampling to get a really precise output that's the same size as the input because you've lost a lot of the spatial structure. You've lost a lot of the information about the fine-grained details in that spatial structure.

And so an alternative would be to take an approach to doing image-to-image, like semantic segmentation, where you don't lose any spatial structure. So here you do convolution, ReLU, convolution, Softmax, is a very, very simple version of that. But every single one of these inputs, because you're keeping the entire dimension, this can be really computationally expensive. So the size of this model would be really huge.

So to bridge and get the best of both worlds, where you're keeping a lot of the high-resolution structure but you're also building models that are computationally efficient and that are forced to learn something that's kind of semantically meaningful is the U-net. So probably many of you have seen this or different variations of this.

But the big intuition here is we have what are called these skip connections across the different encoder and decoder structures of this architecture. So you actually have a connection that takes whatever the size is, and it just takes that via the identity and lets it be skipped over towards the output.

And what this means is the model is not forced to explicitly try to shove a bunch of fine-grained spatial detail into some very low rank, non-spatially dimensional embedding, which is somewhat impossible. It's able to learn the semantic details through this information bottleneck, but then it also gets a lot of the spatial detail through those skipped connections.

And this is really powerful. U-net is a really old architecture, and it is still one of the most widely used architectures for semantic segmentation. It's used all the time in things like remote sensing, like land cover mapping, for example.

And this idea of a skipped connection, this actually is a really useful computational tool. And it's also one of the fundamental underpinnings of what we call the ResNet architecture. So ResNet was invented by Kaiming He, who is also a professor now here at MIT. And this architecture is amazing. We still use this all the time, and it's like 10 years old at this point.

But really, one of the important things that was recognized is essentially-- so one cool intuition about ResNet is every single layer has this skipped connection. So it has this identity pathway that lets you choose whether you want to run the input data through that layer or just skip it, just take this identity mapping and just pass it through to the next layer.

So one kind of cool intuition about this is this actually means that this is a model that can learn something about its architecture. It can learn an optimal depth. Because you can essentially say, I want to just pass this through maybe three of these layers. And then the rest of the time, I'm just going to take these skip connections out to the output. And so then the model itself can implicitly learn the optimal depth for the task that you want to learn.

So one kind of cool intuition, I think, about resonance is that they give the model the flexibility to learn something optimal about the architecture without needing to test out explicitly training a bunch of different architectures side by side. And these residual connections essentially are just x out equals function of x in, so mapping it through that layer plus the input dimension itself.

And if you want to actually change the dimensionality, so you want to have something like downsampling within a ResNet, then you can also have another set of linear weights that downsamples in that skip connection. Cool. And this is a really, really powerful tool. Yeah?

AUDIENCE: So with these skip connections, is the skip becoming another channel that's also like an output channel in the middle, or is it like actually choosing one or the other/

SARA BEERY: So it's kind of choosing one or the other. So the residual connection is essentially-- you assume that the output of your layer and the residual connection are the same size. And then you're just adding them together. But there's different implementations of this that will have some of learned weight.

So, OK, maybe you learn to completely ignore that identity. That same type of skipped connection is also something that gets surfaced via self-attention in transformers, which we'll talk about more in the transformer architecture. But essentially, just this ability to learn how to combine information is valuable.

If it is just an addition or a concatenation or however you want to do that skip connection, one of the cool things about linear layers or convolutional layers, which are constrained linear layers, is essentially even if you just add them together, there's some version of the next layer that could basically learn how to ignore certain dimensions, that input.

But yeah. Like, for example, here, if you do actually want to change the dimensionality, that W layer, that could be learned to be all zeros. So you could just not keep any of that at all if that's what the model wants to learn. Cool.

So far, we've talked about two-dimensional or three-dimensional grids. But one of the things that, of course, comes up all the time in the real world are additional dimensionalities. One really simple one is time.

So here, if you think about convolutions in time, so maybe one version of this is you actually have just a single scalar valued input, maybe something like temperature at a temperature sensor. And then you can think about doing convolutions over time. So instead of doing convolutions over a spatial dimension, you could take convolutions over a temporal dimension and build some representation with time as one of the dimensions of that convolution.

And now if you actually have something like video-- so, for example, now you actually have an image with red, green, and blue channels. And then you have these different frames of an image over time. You can think about building like what we'll call another tensor representation or this block representation where you have the spatial extent.

And here we're now using that third channel to represent time. But then you also have RGB for each of these. So it's actually a four dimensional input. And then you can think of building these 3D convolutions that actually convolve over space and time.

And then your output would be maybe a lower dimensional representation that would actually have those space and time dimensions for each of your tensors. And this is a type of architecture. This type of 3D convolutions is often used when you're trying to do something like action recognition in video.

Now, one thing that this gives you is it makes you shift invariant in time. But you can imagine actually that for some types of recognition tasks, you don't want to be shift invariant in time. You don't want like picking a cup up to be the same action as putting a cup down.

And I mean, they're-- ignoring the motion between, if you had a really low frame rate, if you had this frame and that frame, and then maybe you had that frame and this frame, you want the fact that one happened before the other to matter. That translation invariance is not something you always want.

And so if you don't want to be shift invariant, there's a couple of different approaches that we can use. So one, use an architecture that's not shift invariant, so something like an MLP. There the position in time would actually matter. But like we said, these are it's quite difficult to get these MLPs to learn really complex functions of that almost infinite data.

So another option, and one that is really shown to be a winner, is this idea of adding location information to the input to the convolutional filters through something that we call a positional encoding. So here what we do is you have your input signal.

And then you also have this separate positional encoding, which is something you construct. This is not something that's learned. It's actually constructed and added to the input. And you learn an additional weight that incorporates that positional encoding as part of your convolution. And so there you're doing the same thing that you might with a normal convolutional layer, but you're adding this additional information from position.

And so here we might want to introduce the concept of what gets sometimes called a neural field. So a field from physics, it can be a varying physical quantity of both spatial and temporal coordinates. And a neural field is a field that is parameterized maybe fully or in part by a neural network.

So there's different ways that you can think about doing this and representing these neural fields. But one way to think about it-- so SIREN is an example of one of these neural fields, that model we talked about earlier.

This is a convolutional neural network that's applied per pixel to map from a coordinate grid, so now the input is the positional encoding in an image, to a color. So you're actually learning to model a single image by learning a functional mapping from a position in the image to a color.

And the way that they're doing that is through these sinusoidal things. And so here what you actually do is you take-- the input is actually just the position, so in some coordinate space, and these could be explicit. They could be batched up. Or they also could be continuous because the idea is that this is something that's mapping from some of position to an output.

So now you run these inputs through a model that's just taking x and y. And then you're outputting the color of that value. And this can take continuous coordinates as input, which means that we can then do something like think about learning these functional representations of input space as opposed to only gridwise representations.

And this idea is the fundamental underpinning of quite a popular architecture recently called NeRF, which is essentially a convolutional network applied to a map from a five-dimensional coordinate grid, which is a position and direction of the camera relative to the scene.

And then it gives you an output of color and volumetric density. And this allows us to then actually generate what that same object would look like from different positions than what was in the original training data.

So what that actually looks like is you can take some images of a scene and then generate these essentially walkthroughs of scenes. And these have actually gotten better and better over time. And one of the ways that this is really-- one of the fundamental underpinnings of this idea of building these appropriate positional encodings. Cool.

NeRFs are really fun to play with. Recently, some people at Woods Hole made a NeRF for under the sea that handles the way that light bends under water. Very neat.

So concluding remarks here. Convolution is really just a fundamental operation for image processing or actually for processing anything that has some structure. So this might be something like time or even time, space, and multiple input dimensions.

So say you wanted to use convolution over hyperspectral satellite imagery taken once a week from the entire Earth. Essentially, what it means is you're just chopping up the image into patches, and then you're applying the same function to each patch, but you're doing it through this slide and sum operation.

And this concept appears in basically almost all modern architectures. It appears in CNNs, but also things like transformers, though then the patch-wise operation isn't necessarily convolution, use attention instead, things like NeRFs and more.

So, today, we talked about building better architectures and why. We talked about convolutional layers, pyramids, and receptive fields. We went through at a very high level some of these different ideas that built out different types of architectures.

And then we talked about neural fields and positional encodings, though we'll talk a lot more about positional encodings in the transformers lecture. Any final questions? And if you are going to leave during questions, just be quiet so that we can address them. Yeah?

AUDIENCE: Could you use NeRFs or upscaling?

SARA BEERY: Could you use NeRFs for upscaling? Yes.

AUDIENCE: [INAUDIBLE]

SARA BEERY: Yeah, yeah. So one of the ways that people have used NeRFs is to increase the resolution of, let's say, like a point cloud representation, not exactly like the original NeRFs architecture, but there's been adaptations of it that try to increase spatial resolution. Yeah.

Any other questions? Yes? And then quiet, please, you guys. Thank you so much.

AUDIENCE: [INAUDIBLE]

SARA BEERY: Yeah. So something like strided convolutions, that's actually explicitly doing some form of pooling and downsampling together. So one way you could think about it is like pooling is essentially the same as running a convolutional layer.

But now you're not learning weights. You're just taking, for example, like the maximum function or that average function. That's like that's the function that's applied on that local patch. So essentially, it's a constrained version, a nonlearned version of a convolutional layer. Yeah.

AUDIENCE: And then downsampling is like [INAUDIBLE]?

SARA BEERY: Yeah. Yes?

AUDIENCE: Could you explain a little bit on the neural fields? [INAUDIBLE]?

SARA BEERY: Yeah. So the way that you train it is you take a bunch of different images of a scene from different positions. So often, the training data for NeRFs will be someone taking a video, like walking around an object, for example. And now you have a bunch of examples where now you're going to estimate the relative position to the scene.

And then now you can estimate the position and the direction from the input image. Now for any given point, that's your training data. And the output is the color of the image at that pixel.

AUDIENCE: Is that generalizable, or is just used for picture to video only used for that?

SARA BEERY: It can be used for a lot of different things. So I will say that only just recently, like last week, I saw the first kind of large-scale paper that was taking this concept of NeRF, which gives you this creation of video that you can interpolate between, and then actually trying to move that directly to something that was 3D modeling the scene.

Because the output of NeRF is not a 3D model of the scene. It's what the image would look like taken from a different direction, which is not the same as a 3D model of the scene.

And, actually, a lot of real-world applications we would like to actually be able to use sparse monocular image representations to get out a robust 3D model of the scene. And so they've just started to bring those two things together in recent work from Ben Mildenhall, I think, was the person.

I will say these are also really not super robust. You need a lot of input data. So classical NeRF, like if you only have a few sparse images of a scene, it can be really hard to fit a NeRF appropriately.

And the other thing is it works really well with stuff that is not moving. But if you try and do it with something like a waving tree because the actual thing itself won't-- the math doesn't necessarily optimize because the thing that you're taking images of is changing position over time. That makes it really hard to fit these models. Yeah.

AUDIENCE: Thank you so much.

SARA BEERY: Yeah. Yes?

AUDIENCE: What about problems where the input is not just a picture, but a picture and constants like scalar numbers? [INAUDIBLE] contribute to how the picture should be understood.

SARA BEERY: So maybe like additional metadata. So for example, I might have a picture and a GPS location where the picture was taken. So there's a lot of different ways that we try to incorporate that side information into these architectures. And you can see lots of different examples.

But some really obvious stuff is you can just run your image through that first convolutional layer, and then you can have a separate type of convolution or representation of that positional encoding. And you can either concatenate it to the entire input image. You can introduce it slightly later in the network. There's a lot of different ways you can append additional information into the network during training. Yeah. Yeah. Cool.

Thanks, guys. Oh yeah?

AUDIENCE: When you say there's [INAUDIBLE] video, can you just add additional channels for each frame versus just doing a 3D convolution [INAUDIBLE]?

SARA BEERY: Is there a big difference. So if you add if you add channels for all of the different frames in the video, essentially, that is 3D convolution. Yeah, because you're taking those channels and you're maybe running your patch-wise operation almost like a sliding window over time.

AUDIENCE: Yeah, I guess like you don't add another [INAUDIBLE].

SARA BEERY: So sorry, I'm a bit confused. So if you have an image model that's taking an RGB and height and width, now you take that image model and now you're just saying like stack it over the different frames. How would you not add another dimension if you have a time dimension?

AUDIENCE: Like RGB channels, so you just add more RGB [INAUDIBLE] for each frame.

SARA BEERY: Oh, I see. You could do that. It would get really expensive. And I think one of the things that we definitely see with things like video-- very rarely are those convolutions in the temporal space worth it computationally.

So if you actually look at a lot of the state of the art models for video, often, they are downsampling in the temporal space because you can actually recognize, like most actions through what we call maybe a bag of images. So you don't need every single image. And the difference between the individual frames is really small.

And so like Alyosha Efros has a nice paper where they talk about the complexity of actions by essentially taking any action recognition problem with a video clip and then using the number of frames from that video clip that you need to be able to accurately recognize that action as a representation of the complexity of the problem. And most actions are it's a single frame. You can recognize it from a single frame.

So because of the computational complexity and the fact that often we want these models to maybe run in real time, a lot of times the state-of-the-art models for video are doing something that we call fast and slow, where they will have a really low-dimensional subsampling spatially and maybe even also temporally that's running more frequently and then some of secondary channel that's taking high-resolution inputs, so maybe the full image frame size or something quite close to it.

But it's only doing that periodically. And either it's doing it periodically kind of in some pre-defined way, or it's actually deciding when to run an image through that high-resolution channel.

So they'll take this two-pronged approach where one model is maybe modeling something like the motion and the other one is modeling some fine-grained semantic details.

AUDIENCE: Cool. Thank you.

SARA BEERY: Yeah, video is fascinating and quite complicated and expensive, expensive to store, expensive to train models on, expensive to run inference on. Yeah, it's a mess.