[SQUEAKING]

[RUSTLING]

[CLICKING]

**PHILLIP ISOLA:** Great. So welcome. This lecture is going to be on generalization. So this is the next topic in the series of how do you approximate functions? Now do they actually generalize? And the basic question is, do neural networks generalize? And well, we'll see, yes, the answer is yes to some degree. And then the question is why? Why do they generalize?

A few logistical reminders. Problem set one is due today. Problem set two goes out today. Problem set two will be posted sometime after this class. So we always will post problem sets after the lecture. And you might want to look at the first question of problem set two if you happen to have time before Thursday's lecture, because it can help motivate-- you don't have to solve it, but it can help motivate and put into context what we'll see on Thursday. So I'd recommend taking a look at that first problem. And then you'll kind of know what to look out for in the lecture to help you solve that problem. So just a little piece of advice.

And I think that's it. So we're going to go right in. So the outline for today will be, what is generalization? That should be review. You've all taken machine learning or classes like this before. Then, the next question is, do deep nets generalize? Then, we'll see that there's some classical theories of generalization, and they don't actually work very well for deep learning.

So we'll see some failures of classical generalization theory at explaining phenomena in deep learning. And then I've already given away the answer-- that, to some degree, deep nets, will argue, do generalize. So why do they generalize? What does this tell us about their inductive biases and preferences toward solutions that will generalize over solutions that simply fit the data but don't generalize?

So that's the outline. And I should also say that the material in this lecture is not really textbook material. We gave a few readings, but there are kind of recent papers, so we just listed them as optional. This is very much ongoing science. This is, to me, maybe kind of the biggest, or one of the biggest open sets of questions of topics of inquiry in the field of deep learning and in AI in general, is exactly what are the generalization properties of deep networks?

They seem to work a lot better than we would have expected from classical theory. So why? We don't actually know. We don't have all the answers. This lecture will just pose some questions and give some initial threads that you could pull on. But this is going to be a topic that will change a lot over the next coming years. I think this will develop a lot.

So let's go over the basics of approximation versus generalization. We've so far talked in this course really only about approximation. So approximation is, I have some training data. I want to ask how well can I fit that training data. And we measure the fit with the empirical risk, which is the first equation up here. So we just say, what is the loss L incurred over all of my training data points?

This is a supervised setting where I have data points x and outputs y. I'm trying to predict the correct output y for a given x. So I just sum up all of my prediction errors according to my model f and my loss function L. So this is just, on average, what is my prediction error. And we had a few lectures, and in the problems that you worked out, cases where you can find that class of functions can minimize that towards 0 with sufficient capacity, with sufficient width or depth.

But that's not actually what we care about in machine learning. All we actually care about is the test error, which can be called the population risk. So the population risk is, on new samples from the world, new samples from the data generating process p. So new samples, x, y, new observations, how well do I do in expectation?

And this is like the population, because it's over the entire population of observations I could get from my world's data generating process p. But you could also have an empirical test risk, which would just be over a sample, like my test set is a sample of 10,000 images-- how well do I do on them?

So we have three different kind of topics that we can factorize machine learning into. One is, how well can you approximate the training data? And that's what we've talked about so far. In my family of functions I'm considering parameterized by theta, what is the best theta star that will achieve the lowest empirical risk? And we've said that you can get it to go to 0 under some conditions.

So we can fit our training data. But then the next question is optimization. How do we actually find that theta star? And we've talked a little bit about that. We saw backprop. And on Thursday's lecture, we'll talk more about optimization. But we said that you can often find theta star to some degree.

And this lecture is, well, wait a second. If you find that really good parameter vector on the training data, what is the population risk? What is the risk on the test data? So what is the gap between the training error and the test error? And that's the question of generalization. And that's all we actually care about because we want to do well when we deploy the system in the world and we see new observations. This should be review. This is like ML 101. Yeah, question?

**AUDIENCE:** If the r in approximations supposed to have a hat on it, it's the question about training data?

**PHILLIP ISOLA:** Yeah, I think you could ask the question-- so an approximation-- yeah, typically, it would be, can I minimize the empirical risk on the training data? Yes. Oh, I see. Yeah, we could have denoted that with a hat. But I suppose you could also ask the question, is the best possible function in the hypothesis space going to achieve low risk on the population as well? So you could ask-- both of those are questions of approximation.

So let's go over some intuitive ideas about generalization just to get warmed up and get you thinking about this. So here's a bad data set. Suppose we have a data set which is just a bunch of photos of cats, and we want to train a cat versus dog classifier. So clearly, this is not going to work. You only observe cats. You're not going to get a cat versus dog classifier. So what's wrong with this data? What would make for better training data for a cat versus dog classifier? Some obvious ideas. Yeah?

**AUDIENCE:** [INAUDIBLE]

**PHILLIP ISOLA:** You want to have some dogs. Yeah, what else? What if you have just one cat, one dog, and then a lot of copies of that image? You want diversity. You want coverage. You want to show all your different conditions. So there are certain properties of the data that are critical. You can't really understand generalization without understanding the data distribution that you're training on.

And if your training set doesn't represent all the interesting aspects of variation that you'll encounter in the world, then you're not going to generalize well. So data is critical to generalization. And we have to understand data to understand generalization. The other thing that's critical to understanding generalization is the model, which will be most of what we focus on. I just wanted to start by saying that the data is also a critical part of the recipe for generalization.

So here's a bad model-- the filing cabinet. So the filing cabinet model is going to be this little Python program here, where we have a dictionary called cabinet. And if we see a new data point x, we put it into our cabinet along with the correct prediction y for that x. And if we come across that x again, we just hash into that cabinet. We just pull out that file. We see what the answer is. So this is sometimes called memorization. I'm just memorizing my training data. But the questions for you are, well, what will the approximation error be of the filing cabinet model? Yeah?

**AUDIENCE:** 0.

**PHILLIP ISOLA:** 0-- right. So the approximation error is zero because for every data point that I train on, I can retrieve the exact memorized answer. What about the generalization error? What will the generalization error of the filing cabinet be? Generalization error is the difference between the empirical risk and the population risk. So roughly, what will it be? Will it be good, will it be bad?

**AUDIENCE:** Bad.

**PHILLIP ISOLA:** Bad. Yeah, it'll generally be bad unless it happens to be that the true function we're modeling is 0 almost everywhere, which would be a very strange function. Because we said by-- if I don't have my file in the cabinet, I will just not know what to do. But maybe I got lucky, and the true function is 0 everywhere. So you could have a filing cabinet that gets low generalization error, but it would be a very strange function that you're modeling in that case.

So we'll come back to the filing cabinet model. And one of the questions is, are deep nets like filing cabinets or something else? Do they just memorize and not generalize, or do they do something else? Here's another bad model just, I guess, a little bit for fun. So this is Paul the Octopus. And Paul can decide who's going to win a soccer match between two different countries.

And Paul was super accurate, and people would make their bets based on what Paul said. And what Paul did is just went into the tank of the flag that it thought would win the match. And it was like it got 6 out of 6 correct predictions. So would you trust Paul to make the correct prediction next year?

Yeah, it's probably like some super intelligent octopus. No, obviously, you wouldn't trust Paul. It must have just generalized-- sorry, it fit the training data well. It's not a filing cabinet. It is making some decision, but it happened to get lucky. It's like some random function that happened to fit the training data, but you'd never expect it to actually have a true model of soccer. So you can have functions that get lucky and make interesting predictions out of sample on the test data, but wrong predictions. And Paul is an example.

So our big question in this course, in this lecture, is do deep nets generalize? So here's a simple warm-up. Here's the filing cabinet fit to these black data points. So my training data is the black points. It's a scalar to scalar mapping. I'm predicting y given x. And the red line is the fit of my model.

And if my model is a filing cabinet, what it's saying is that I've memorized that if I see one of my training data points, I know exactly what to predict. That's the kind of red lines, like the delta functions going down to each of those dots, or up to the dots. And everywhere else, I'll predict 0.

So anything that is not anything on the x-axis-- any point I could sample on the x-axis that's not in my training set that's not where the black dots lie, I will predict 0 for. So this is a very bad kind of generalization. So here's the same data, where I've just fit it with a three layer ReLU MLP.

Not too surprising. A ReLU MLP can fit data, but notice that it fits it in a very different way. So both of these achieve 0 approximation error. Both of them have fit the training data. But if that were all that mattered, then we would have to say these are equally good. But that's not all that matters. It also matters how you interpolate and extrapolate from the training data.

And deep nets do that in a way that's markedly different than a filing cabinet. So memorization is just, what do I predict on the training data points? And filing cabinets can memorize-- deep nets can memorize. And generalization is, what do I do on the points where I didn't have training data? And we might expect that this type of interpolation, this type of generalization is better than that other type, because most functions in the world are going to be smooth as opposed to spiky. You could imagine an adversarial world where that's not the case, but our world looks more like this on the right.

So that's a very simple deep net. But the real feats of generalization that we get excited about these days are with fancy, big modern nets. So Here's a little experiment you can do. I just ran this for myself this weekend, but I think it might be a fun thing for you to play with or maybe take into a final project. So let's try to ask, are modern deep nets-- the best model you can think of-- are they acting like a filing cabinet or are they doing something more?

It's a little bit hard to say, because if you train on enough data-- and these are trained on billions or trillions of data points-- if you trained on enough data and you just memorize the data, well, your test conditions might look identical to your-- if I've trained on infinite things, then my test cases are just the same as my training data. So are they close to that-- just memorizing nearly infinite things, or are they not?

So LLM-- if you don't know, this is Large Language Model-- ChatGPT, latest greatest deep nets are in that family. So big model-- so is it acting like a filing cabinet? Can we actually measure if it's generalizing or not? So here's the question we can ask. How big would the filing cabinet have to be in order to do the feats of prediction that ChatGPT or some big LLM can do?

So remember our model of the filing cabinet is just, we return 0 if we have a data point that we've never seen before. So here's just a simple counting experiment. I think you can make more sophisticated versions to this, but I found it fun. So we'll sample, a sequence-- so language models are deep nets that take sequences of words as input and make some prediction from them.

So we'll sample a sequence of n words, and we'll be sampling from a vocab of size m. So we have m nouns-- 1,000 different nouns, and we make a sentence of length-- or a set of a sequence of nouns of length 5. So then n would be 5. So how many unique input sequences are there?

So just simple combinatorics, m to the n. So I have m possible words, and I have n of them in a row. I sample independently each one. That's the number of such sequences. So now, we can ask, well, if we take a random such sequence and we input it into our model, did the model make a reasonable prediction?

So technically, if this is our model of the filing cabinet, we're just asking, did it make a non-zero prediction? But did it output a correct prediction? So we can take our favorite language model. We can ask our favorite question about the input sequence and have the language model try to solve that question That's how they work. You can prompt them to answer different questions about your input sequence. We'll ask how often is the output-- something that's non-trivial-- not what a filing cabinet would do.

And then we can say that the filing cabinet, in order to get a score of s, equals the number of unique sequences times the percent of times those sequences mapped to something non-trivial. It would have to be that big. It'd have to be taking that many unique sequences and achieving this proportion of correct predictions would have to have that many cabinets. And if that number is really, really big-- bigger than we think language models are, or bigger than the amount of data they've been trained on, then we know that language models are not doing pure memorization.

So you can make up your own version of this, but just to convince myself, I ran it like this. I took 30 fruit names-- apple, orange, pear, whatever. So my vocab is size 30, and my sequence length is size 10. So I'm going to sample 10 random fruits from that list. So there's 10 to the 30th possible input sequences.

And then I'm going to have my language model answer this question about those sequences. So we'll talk more about language models and prompting later. But just you've all done this with things like ChatGPT or Claude and so forth. You can say how many citrus fruits-- this is just my toy problem I'm giving it to try to estimate these probabilities.

How many citrus fruits are in this list? And I give it this random sequence of 10 fruits sampled from that vocab of size 30. And it gets it correct. And so I did this 5 times. I'm not saying this is like the best experiment, but it got 5 out of 5, so it's correct every single time. So my estimate for the probability of being correct is 1.

So now, we can just estimate-- you have p times m to the n is going to be 100 trillion. So the filing cabinet would have to have 100 trillion files. Also the training data-- the way the filing cabinet works is it only inputs a new data point into the cabinet if it sees that data point in the training set. So it means the training set has to have 100 million examples for that model-- sorry, 100 trillion. And we think that ChatGPT and things like this are trained on more like tens of trillions. So I think I exceeded that. But you could always make this list longer and run the same experiment-- get it to a quadrillion, and really convince yourself that these things are not just memorizing.

Another just for fun one. We could do the same thing, but we can use a text-to-image model, and now have it draw the fruits, and ask how many unique different-- how often will it be correct in drawing the fruits? So I asked that to-- ChatGPT can do both of these things now. These are kind of multimodal models. So you can say draw these fruits. I sample a random sequence of 10 fruits and I check if it's correct. Is this one correct?

**AUDIENCE:** No.

**PHILLIP ISOLA:** I said no. It's almost correct. I think it got 9 out of 10. I don't think it has a passion fruit. But anyway, I don't know what passion fruits look like. But I called this one incorrect. And so I estimated--

**AUDIENCE:** [INAUDIBLE]

**PHILLIP ISOLA:** Does have another wrong thing? OK. Yeah?

**AUDIENCE:** [INAUDIBLE]

**PHILLIP ISOLA:** Cantaloupe or--

**AUDIENCE:** Yeah.

**AUDIENCE:** Coconut as well.

**PHILLIP ISOLA:** Oh, yeah, coconuts missing. Maybe that was a passion fruit up there. But anyway, this one's wrong. But I ran this eight times, and one out of the eight times I thought it got it correct. I really saw each of the fruits represented. So now, our proportion p is 0.125, 1 over 8.

And that means that you'd have to have a filing cabinet of size 12 trillion. So I don't think this has been 12 trillion images, but it's hard to know. OpenAI is secretive. But you can do these types of experiments, and I think you can pretty quickly convince yourself that the way that modern deep nets are working is not like a filing cabinet. It has some properties in common with the filing cabinet. We'll come back to that, but that doesn't fully explain what they're doing.

Yeah, and so just a suggestion for final project-- I think it'd be fun to really do that systematically and rigorously, and try to actually estimate how many unique concepts these things have. I haven't really seen very many papers that do this type of experiment, but I think it'd be fun.

So I got kind of excited about the feats of generalization of deep nets in this project from a few years ago. And so what we did here is we trained a deep net to take as input sketches and produces output photos of cats in this case. So a neural net just maps. And we'll talk about these generative models and image-to-image models a little bit later in the course. But for now, it's just a mapping from x to y-- it happens to be sketches to photos.

And so what we did in this project is we didn't use human hand-drawn sketches because those were expensive to acquire. So instead, we just use this thing called HED, which is like an edge detector. And it's like a proxy for what a human might have drawn, but it doesn't quite match what a human would really draw. So from a kind of naive machine learning perspective, you would say, if I train this on a ton of examples of edge maps and cats, and I give it a new edge map, it should predict a new cat because this is just generalization to independently and identically sampled data points from the same population. So that's like classical generalization theory in machine learning.

But the really interesting thing is it also worked when you give it hand-drawn cats which it had never been trained on. So these are inputs that are actually out of distribution-- systematically different than what it was trained on. So in the prehistory of deep learning, this was the popular thing of the day. Now, of course we've seen much more exciting things. But if I sketch a cat, it doesn't look anything like an edge map. And it just works, and it's still cool even today. So people had a lot of fun.

[LAUGHTER]

And this is generalization. It's never been trained on these types of things. It has some kind of creativity almost. We have to be careful not to anthropomorphize too much, but there's something here which is a little bit magical and not fully understood. But I think we can start to get a sense of what's going on with an experiment like this.

So I'm just trying to build up intuitions-- do simple kind of experiments. We'll get into some more technical theory in a bit. But here we go. I just draw two eyes. Now, what if I draw three eyes? What's going to happen? It's never seen a photo of a cat with three eyes. How would it know what to do? But it works. It just puts that eye where that eye was drawn. A lot of eyes. You can add the body. That makes sense.

So how is it doing this? How is it generalizing to more eyes than it ever saw at training time? And I think there's not a single explanation, but the best one I have in mind is it's the inductive biases of convolutional nets. So this network was a ConvNet. We've seen that.

What do ConvNets do? They take an input, they chop it up into patches, and then they independently and identically process every patch. So they're factorizing the problem into a lot of independent decisions. And when you decompose something into these independent components, you get what's called compositionality, where you can take a new composition of things and factor them into these independent components, these patches.

And it should it should know what to do, because it only ever has to imitate the training data on these patches. And then the combination of patches is just the rule of stitch them together. So for every single patch, it's doing-- in distribution generalization, it's seen a lot of things that look like ovals, and they match eyes. And then it can generalize to more eyes because I've just factorized the problem in this simple way.

So architectural constraints and biases and the way that we enforce symmetries and factorization into our models allows you to do certain kinds of feats of generalization. And we saw that with the ability of graph nets to generalize to new permutations of their inputs, because they're permutation-invariant or equivariant. So that's just built into the architecture.

I don't have to have seen every permutation. I can just have seen some permutations, and then it will generalize to other permutations. That's baked into the architecture. It's not something you have to learn from the data. So architecture is one of the main levers we have for these feats of generalization that violate just statistical learning theory, or that go beyond just naive statistical learning theory.

So let's get a little bit into generalization theory and start with the classic picture. So most of generalization theory comes from Occam. It comes from the Occam's razor idea. It's all variations on Occam's razor in some sense. So just to remind you, Occam's razor says among competing explanations that fit the data equally well, you should prefer the simplest And the generalization way of stating that is that the simplest model that fits the data will be most likely to be the true model. Therefore, it will be most likely to generalize the best.

Just one little note is that I think sometimes people forget to add the "fits the data." So it's not true the simplest model is most likely to be true, or it's most likely to generalize best. It's the simplest model that fits the data that actually explains the data.

You need that, otherwise you get these absurd statements like, oh, well, I should use a linear model. But no, it doesn't fit the data. So you have to use a complex enough model that fits the data, but then you want to prefer the simpler one. And there's a lot of deep arguments for why that should be the case. It's intuitively clear from the idea of Occam's razor that we learn in grade school or whenever we learn it. But also there's deep mathematics that can be used to derive formal versions of this statement. Yeah, question?

**AUDIENCE:** If someone were to compare this [INAUDIBLE] that all models [INAUDIBLE]. How would you compare this [INAUDIBLE]?

**PHILLIP ISOLA:** The question-- that's really interesting. There's another quote-- "all models are wrong, but some are useful." Yeah, that's intriguing to think about. I would say that the "all models" is-- so this is only the set of models that are in the equivalence class of all equally well fitting the data.

So we're not really considering models that are wrong in terms of only approximating the data. So it's a slightly different setting that we're looking at there. But yeah, I don't know. There's probably more discussion around that, but I don't have a crisp answer right now. Yeah, question?

**AUDIENCE:** Has this statement been proven rigorously or is it a heuristic?

**PHILLIP ISOLA:** So I would say in this form-- so the question-- has this statement been proven rigorously, or is it a heuristic? In this form, it's a heuristic because how do we measure simplest? How do we measure generalization? We need to make these mathematically precise to make it rigorous. But there are rigorous versions of the Occam's razor argument.

My favorite one is this one from various people, including Solomonoff, which is that the shortest program that perfectly fits or generates the data-- that just outputs the data set-- is the one that will generalize best and make the best predictions. There's an equivalent version of this statement, which is that the most compressed representation of your data is the best model of your data. There's some equivalency between these two things.

So I'm not going to get into all the technical details of that. But this is the place to start reading if you want mathematically precise arguments for Occam's razor. And the problem, though, is that finding the shortest program that fits the data requires searching over the space of all possible programs. Imagine searching over the space of all possible Python programs. You just can't do that.

And deep learning doesn't search over the space of all possible programs. It just searches over the space of all weights and biases of a particular model architecture. So this is completely intractable, but it comes up in algorithmic information theory and areas like that. So it's good to keep in mind, this is kind of like the mathematical basis for generalization theory. But actually finding the optimal program or the shortest program is intractable, so we're not going to quite do that.

But this is what we want. We want a way of measuring simplicity or complexity of a hypothesis space or of a solution to a problem. And this is kind of the best notion of simplicity, but we can't quite use it. So what else can we use? So let's do a quick review of overfitting-- the idea of overfitting and the bias variance trade-off.

So remember that the test error is equal to the train error plus the difference between the train error and the test error. And we call the training error the bias and the difference between the test error and the train error the variance. And the kind of classical machine learning 101-- picture of how things work-- is that as I increase the kind of capacity, the number of functions under consideration in my hypothesis space that I'm searching over, as I make bigger and bigger and bigger deep nets with more and more and more parameters, then eventually I'll get universal approximation.

If I get width goes to infinity, I can fit any function. So I can minimize my loss on the training error, so the training risk goes down. But I'll start overfitting, and the test risk will eventually go up, because I'm finding fits to the data which are kind of fitting to properties of the data that don't actually generalize, like noise, or the fact that I got this particular batch of samples as opposed to a different batch of samples. So this is the overfitting scenario that you have likely seen before.

And one of the ways of measuring model capacity-- modeling the x-axis or measuring the complexity of a hypothesis space is via number of parameters. And this is one of the classical ways that you might have encountered. If I want to find my best machine learning model, I need to control the number of parameters. If I have too many parameters, it will overfit.

So let's look at what happens if I'm doing some kind of polynomial regression onto these red points here. So the orange line is my fit to the data. And if I use a degree 1 polynomial-- I think it's not quite polynomials. It's a particular family of polynomials. I'm finding the coefficients of a weighted sum of a particular family of simple functions, like Legendre polynomials, I think, in this case. But it doesn't matter too much.

So I'm going to say with d equals 1 is the degree of that family of polynomials. It's just d equals 1 means linear, and a linear fit can't do a great job. The truth function is the blue line, and the training data is the red points. What happens as I increase the wiggliness of the polynomials in my family? I include more and more polynomials that are higher and higher frequency and the kind of wiggles that they can express.

So if I go up to a quadratic or cubic model in this case-- well, the ground truth function happened to be cubic or happened to have order 3 Legendre polynomial, and I get a really good fit. And then overfitting is, if I go up to-- have too many degrees of freedom, I can actually not only recover the true data, but I'm also fitting to the noise. This deviation from the blue line is just noise. And the model is wildly deviating in order to capture that aspect that doesn't actually generalize. It doesn't actually tell them something about the true underlying function.

So this is our classic picture of overfitting, but here's the surprise if you haven't seen it before. What happens if I go not to d equals 20, but d equals 1,000? I have a really, really expressive class of polynomials that can fit really, really crazy wiggly functions. Is it going to go more crazy and overfit even worse in the sense that these deviations will just go to infinity, or will it do something different? What do you think? Some of you probably have seen this before, but let's go up here. Yeah?

**AUDIENCE:** I mean, you should have a large manifold of fits that all fit perfectly, right?

**PHILLIP ISOLA:** Yes.

**AUDIENCE:** So some of them could be good, some of them could be bad.

**PHILLIP ISOLA:** Yeah, so the answer to-- and that's a quite sophisticated answer-- is you should have a large set of functions that all perfectly fit the data. And so some of them should be good and some of them should be bad. But which one will you arrive at? Well, it's going to depend on other factors. It's going to depend on how you're searching over that space, or what optimizer are you using? How are you minimizing your training error? It's going to depend on other factors beyond just randomly selecting one of the solutions that fits the data.

So here's what happens for this regression problem with these polynomials. So when d equals 1,000, you actually are able to fit this really spiky function, but it adheres much more closely to the true smooth solution. And this phenomena is called least-- is called double descent. So it's the idea that overfitting is actually-- the classical picture of overfitting is a little bit inaccurate.

And what actually happens is you start overfitting really badly as you add more parameters. But eventually, if you add enough parameters, for certain types of models, including deep nets and also including these polynomial regression problems, you end up getting this kind of behavior where you start actually getting better generalization error with more parameters.

And the basic hypothesis here is that-- and this is provable for certain simple systems, but it's not really provable for deep nets. But the basic hypothesis for deep nets and the general statement is that the learned model has two components. With enough capacity, it will find a simple fit to the data. And then it will also use little spikes of capacity to fit the outliers.

So there's some kind of simple component, which is predictive and generalizes well to new samples from the population, and a spiking component that overfits noise and things that don't generalize in the training data. So it's like memorization. Spikiness is like memorization. I'm just memorizing those-- when my input is this, I memorized what it was. So it's memorization plus generalization. It's a really interesting combination that, until a few years ago, people didn't widely appreciate.

There's a nice visualization of this with an MLP that you can find at this link. It looks basically the same as what I'm showing you, but not quite as spiky. So deep nets always are a little bit Messier than more classical models, so it's easier to see with this polynomial regression model. Yeah, question?

**AUDIENCE:** So from my understanding, a polynomial regression is a closed form solution. So I guess you could see that this is always true in this case. But for a neural network where you're doing like a stochastic optimization process, I'm wondering how you can be sure you'll always arrive at this function and not, as you said, a function that is wildly inaccurate?

**PHILLIP ISOLA:** Yeah, so the question is, for polynomial regression, there's a closed form solution, and you can really prove that this will be what happens. And that solution will have this property if that's the case. For a neural net, the solution you arrive at will depend on the initialization, which might be random. It will depend on the stochastic mini batches, which are going to be random.

There's a lot of randomness, and it's much harder to make clean statements about what solution you'll arrive at. So instead, we can make empirical statements. We can say that if we run this neural network over and over again with the Adam optimizer, or the SGD optimizer, here's what we observe. And I'll show you that on the next slide. Or we can make theoretical statements like, the distribution of solutions that will be arrived at under certain conditions will be smooth and spiky with high probability, stuff like this. Yep, one more question?

**AUDIENCE:** When would you know that you're using the right computational resources rather than optimizing to a new model? So if it takes 1,000 to make an optimal solution, but it takes a lot of computational effort, should you stop at some point, or turn to other model that will potentially fit better?

**PHILLIP ISOLA:** Yeah, so when should you decide where to stop in an optimization process is the question, or at least stopping. Yeah, so that's a hard question. And so the number of steps of gradient descent that you do is kind of a hyperparameter, and it can affect the solution. And if you do too many, then you might be overfitting. And that's kind of a classical idea.

But what we'll see is that in this neural net era, it tends to be just train forever. And the longer you train, the better you do because there's a double descent phenomenon, which I'll show in the next slide in terms of computation. The longer you train, the better you'll do in this particular family of models is the general principle. But yeah, of course, there's more that can be said about that.

So this was a really cool paper from a few years ago that got people excited. So the double descent phenomenon is just a consequence of what I already showed you. So we have regular classical overfitting. As I increase my capacity of my hypothesis space, which we're right now naively calculating as the number of parameters in the hypothesis space, then my training error goes down, but my test error eventually goes up. I overfit.

But then after overfitting, eventually, I learned the simple function and the spikes, and so I actually start to generalize better again. And there's this thing called interpolation threshold, which is the point at which I can perfectly memorize all the training data. And I think one intuitive-- I'm not sure if this is going to turn out to be precisely true, but my intuitive picture of what's going on is that you're trying to fit the training data, so you have to memorize it. And you have to learn a really crazy function to memorize it.

But then once you add more capacity, where now there are a lot of solutions that equally well memorize the data equally well, interpolate the data, then you can select within all of those set of solutions that fit the data the one that is the smoothest or has other nice properties. And the pressures that select within the set of things that fit the data the one that is the smoothest are sometimes called regularizers, or implicit regularizers, or inductive biases. And we'll talk more about those in a bit.

It's like, for a while, you're just fitting the data. You have to find a crazy fit. And then once you have more capacity, you can search over that kind of degeneracy to find the one that is the best according to other criteria, like smoothness. Yeah, we'll go with a question.

**AUDIENCE:** For [INAUDIBLE] descent, [INAUDIBLE] you should stop at the first place spot, or you should train like [INAUDIBLE] training until the second coverage [INAUDIBLE].

**PHILLIP ISOLA:** Yeah, so the question is, should you stop at the first minima of the double descent curve, where you have low risk on the population, or should you stop at-- or should you just train forever? And generally, I would say that it will depend on how much capacity you have available to you, because this is costly to move out in the x-axis.

Capacity can be measured in parameters, but it could also be measured in flops. So there's kind of double descent also in compute time. And so it will depend on the cost-benefit analysis there. But generally, in deep learning, we're out in this regime. We're past the first peak, and we're in the part where you just train longer, and longer, and longer, and it smoothly will get better. That seems to be the empirical property we have. One more, and then I'll have to move on.

**AUDIENCE:** What's the correlation between doing polynomial regression and training for a higher order polynomial? Is it similar to a Taylor series as that sum approaches infinity that you could approximate any arbitrary function? What's the relationship between that and a polynomial function in that [INAUDIBLE]?

**PHILLIP ISOLA:** What's the relationship between polynomial regression and deep nets-- so roughly, you can think of the last layer of an MLP as a linear combination of some basis functions, of some features. So every neuron on the previous layer is a feature of the data. And those features could be like polynomial functions of a scalar input, and that would be polynomial regression.

Or they could be Fourier. They could be sines and cosines with different frequencies. And in deep learning, they'll be learned functions, but they have the capacity to learn those types of functions if those are the right ones to learn. So here's an empirical result of double descent. And this is training an MNIST classifier. And you can see this really happens.

So the train error just always goes down as I add more weights. So I increase the width of my model. I get universal approximation, but test error spikes and then it goes back down. This stuff is a little finicky to actually observe in practice. So I actually tried to make my own example of this, and I couldn't get it to work.

But if you read the double descent papers, you'll realize that it only really shows up in certain regimes. And you have to make sure not to have momentum in this and that, and weight decay. There's a lot of aspects of the optimization process that can make this disappear. But it is a real phenomena that does exist in the right conditions. So don't worry if you don't see it in your graphs. You might just be out in this regime, or you might have other effects that are dampening out that spike. Yeah, question over here?

**AUDIENCE:** [INAUDIBLE]

**PHILLIP ISOLA:** Yeah, I'll have to think-- I'd have to think a bit more about the interaction of momentum with this. But I would say that, generally, all the tricks that we use in practice are just getting rid of that spike and pushing us more out to the right. So it's not like we're losing that benefit. We're reducing the negative effect of having this spike in the middle.

So one of the interesting things that could help explain this phenomenon is that the more features that you have, that you're regressing on top of-- these could be features, like the hidden units of an MLP, or in this case, these are what are called random Fourier features-- like, random sines and cosines, random periodic functions to the data. As I have more and more features, well-- at first, I underfit the data. And then I start using all the features.

And my weights are assigned to those features spike because I have to make a linear combination that fits the data. I have to be using all of them. But then what happens is, as I add more and more features, the norm of the weight vector, or the parameter norm-- the norm of that vector, the size of that vector actually decays. So I have more features, but the weights I'm applying to those features are getting smaller and smaller in terms of their overall magnitude-- the norm of the entire parameter vector.

So it's like, more features means lower norm parameter solutions, so in some sense kind of smoother in a particular sense. And so this is just saying that counting number of parameters is not the right thing to do. It might be that the parameter norm or some other property of the parameters is what is the more meaningful measure of the complexity of the solution, or how wiggly it is, or how much it deviates from a nice smooth fit to the underlying trend. Yeah, question?

**AUDIENCE:** Did that line up-- could that be lined up with the interpolation threshold?

**PHILLIP ISOLA:** So the question is, does that peak line up with the interpolation threshold where you perfectly fit the data? Yes, it's exactly at the interpolation threshold. And you can show these things for linear models, and it's all provable. But for deep nets, it's just empirical. Let me keep going, and we can come back to more questions later.

So how should we measure complexity? So number of neurons, number of weights, number of parameters? No, that just doesn't work. So double descent shows that, as I increase the number of neurons, the number of parameters, I don't necessarily overfit. So number of parameters is not really the right notion of complexity according to the classical theory that having a more expressive, higher capacity hypothesis space will cause overfitting. So number of parameters, no. Parameter norm-- maybe that seems a little bit better, but it has its own flaws.

Just to emphasize this with maybe a really obvious example-- does the number of parameters a deep net has-- why would we actually expect that to matter? So consider these two deep nets, f and g. And f has a lot of parameters, a lot of edges. And g has only a few parameters, a few edges. And we're going to consider a new function, h, which is just a combination of a tiny, tiny bit of f-- 10 to the negative 100 worth of f. And then almost entirely we're going to use g.

And now, we're going to ask how many parameters would it take to fit h to some epsilon degree of approximation? And the point here is that if I just fit h with g, I would only be using a few parameters, but get within epsilon is 10 to the negative 100 of the correct solution. So it's kind of the count of parameters is not what matters. Something about the size has to also be what matters in terms of how well I'm able to approximate different types of functions. So parameter norm, size of the parameters matters more than count. But parameter norm is also not everything.

So let's try another idea from classical theory, which is-- OK, it's not number of parameters that is the proper measure of the complexity of a hypothesis space or a neural architecture. How about the number of unique functions in the hypothesis space? It's another measure of capacity. And this is related to a theory that Vapnik and Chervonenkis-- I don't know if I got the pronunciation right-- anyway, VC theory.

So remember, generalization error is the difference between the training error and the test error, or the training error and the population error. And the basic argument of this theory says that if the number of training data points I fit to is much larger than the number of functions under consideration in my hypothesis space I'm searching over, then you can guarantee to some degree that the population error matches the training error.

So something about the size of the training data and the size-- the number of distinct functions that I'm searching over-- something about the ratio between these things. So the intuition is like this. So we could define something called a false positive, which would be a function that fits the training data but has high population risk-- a function that just gets lucky. It fits but doesn't generalize.

So what is the chance of getting a false positive if I'm searching over all of the different candidate functions in my function class? So the chance of getting a false positive will be high if we just have more candidate functions. Like, if I just have more candidate functions, there's more random ones that will get lucky-- so that intuition.

And then it should be lower if I have more data points, because every additional data point rules out certain candidate functions from being able to fit the data right. Every new data point is a new constraint. And it says there are fewer functions in my hypothesis space, in my function class that can fit the data. So therefore, if the function class is small and the training data is big, the chance of a false positive, of just getting lucky, is low. And therefore, I have a higher chance that I really found a solution that generalizes. That's kind of the rough argument. And you can make this mathematically precise.

I have one more picture to make this argument just in case that wasn't clear. But let's imagine that I have this very simple setting. I have a hypothesis space, which is a discrete set of candidate functions, these circles. So every circle is a candidate function. The purple circles are the functions that fit my training data. And the true function is in green.

And we're going to make some assumptions. We're going to assume that there's no noise in the training data. We're going to assume that the true function is in this set. So it's not like I have a chance of finding it. And we'll assume that my way of selecting from this set is just I pick at random one of the purple points. And the true function is also one of-- the true function also perfectly fits the training data.

So my optimization algorithm is just pick at random one of the points that fits the training data. So what happens as I increase the training data? As I increase the training data, some of those purple points turn off, because if I add a new data point, some of the functions that fit the data previously no longer will fit the new data because there's more constraints. Every new data point adds new constraints that you might violate.

So as I increase data, I strictly will decrease, or at least not increase, the number of these purple points. So what happened is I add training data. Right now, I have a 1 out of 6 chance of if I randomly sample one of the purple or green points, 1 out of 6 times I will get the true function. But if I add more data now, it's 1 out of 4 times. So adding data will increase my chance of getting the true function under this procedure.

If I reduce the capacity, what does that mean? That just means removing circles. So the capacity is the number of functions in my function class. If I just remove circles at random-- it's not necessarily how you would normally reduce capacity, but this is one model of reducing capacity-- while still making sure that the true function is within my hypothesis space, then I will remove some of the purple spurious fits, and I'll increase the probability of, if I just select a random function, getting the right one. So there's these two things that are happening. Control capacity-- add more data. They'll both end up increasing my probability of getting the true function that generalizes.

So a few more technical details. How do you count the number of functions in a function class? I said we could have a discrete set of functions under consideration. But with a neural net, it's a continuous thing. So it turns out that for the VC theory, it suffices to do the following. You simply count up how many discrete outputs your function can apply to the training data.

So the simple version of this is I'm going to assume that my function is outputting-- it's a binary output. So it's going to output negative 1 or 1. So it's like binary classifications of my data. These are called dichotomies. And let's say I'm doing an image classification system. So I have some number of images.

And one unique dichotomy is plus 1 to the first image, plus 1 to the next image, negative 1 to the thousandth image, and so forth. And I can count up how many unique dichotomies I can apply to my data, and that will be called the VC dimension of that model. Just one sec. So VC dimension is the number of dichotomies over my n training points.

And you can prove, under some assumptions, this generalization bound that the population risk will be bounded by something related to this square root of d over n. So if my model class can represent more dichotomies, can classify the training data in more different ways, then my generalization bound will increase, and I will not be able to say how well I'll do on the test data. It's like, more functions under consideration, so decreasing.

But as I add more data, n, then that will rule out more and more of the spurious fits. And so we have this ratio, and we say that our population error will be bounded by this ratio. So the cartoon argument is the same, but you can prove under some conditions that this is the exact bound that you get. Let me finish this argument, and then we'll go a few questions.

So here's the interesting thing. Neural nets can basically fit any dichotomy over regular data sets. So given a data set of cats versus dogs I'm trying to classify, I can train the neural net, and it will correctly classify cats and dogs. But I can do the following thing. I can randomly shuffle the labels.

So I take my data set where the cats are labeled cat and dogs are labeled dog, and now I'll just randomly permute the labels-- assign each image a random label-- just choose different random ordering. So now, some dogs are labeled cats, some cats are labeled dog, some cats are labeled cats. It's completely random labels.

And it turns out that, in practice, the neural nets we use can fit any random labeling. So in other words, they can achieve any dichotomy. So in other words, the VC dimension is the total possible numbers of dichotomies of n points. So how many different ways can I binarily classify n points? 2 to the n possible ways.

So every single point I can assign either a plus 1 or minus 1, and then I can do that for every other point-- for n points. And so the VC dimension d is equal to 2 to the n. And the generalization bound is exponential in the number of data training points-- is 2 to the n over square root of that. And it turns out that this is just extremely loose. And in practice, it's just vacuous. It says that my population error will be greater than 100% if this is a classifier, or my population error is less than 10 billion percent or something like this.

So as n becomes large, this scales exponentially, and it becomes a vacuous bound. It tells us nothing about how well the system will generalize. So the VC theory doesn't explain deep learning. It has the right, I think, paradigm of thinking about the size of the hypothesis space, the number of constraints, how many functions could fit the data without generalizing well.

But the assumptions it makes are violated and don't take into account the full picture of all of the factors that are affecting deep learning. One of the main ones is that, in deep learning, we're not just picking a random hypothesis that fits the data. We're using gradient descent, and that's going to pick certain hypotheses and prefer certain hypotheses over others. Among all hypotheses that fit the data, it might prefer simple ones over complex ones, which we'll see in a minute.

And this is one of the optional readings. There's a nice paper from a few years ago called "Understanding Deep Learning Requires Rethinking Generalization." It ran that experiment. It took MNIST, and CIFAR, and some of the classic data sets, and it randomly shuffled the labels. And it showed that you can get 100% training accuracy on any random shuffling of the labels.

But if you use the true labels, then your test accuracy will be high. But if you just randomly shuffle the labels, now, there's no ability to generalize at all. Your test accuracy will be low. And VC theory would say that the generalization bound is completely vacuous. It doesn't tell us anything about how well one or the other of these things will generalize to the test data. So there must be something more about what neural nets are doing that's not explained by VC theory.

So neural nets can fit random labels, yet when they're trained on real labels, they generalize. This is the violation of classical theory. Any questions there? I think we had a few, so we can go back to the slides where you had them if you still have them. Yeah?

**AUDIENCE:** So a dichotomy is just [INAUDIBLE].

**PHILLIP ISOLA:** Yeah, dichotomy is a binary labeling of all of the training data points, so a vector of assignments of the training points to 0 or 1-- negative 1 or 1. So how should we measure model complexity in the deep learning era? Should it just be the number of distinct functions the model can represent? No, because deep nets can represent a huge number of functions, and so many functions that the classical bounds on-- that tell us how capacity relates to generalization don't hold.

They would say that this is way too high capacity a model. There must be some other thing. There must be something else that is helping us to find solutions that generalize rather than solutions that don't generalize. And unfortunately, I have to say that for deep learning, we really don't know what it is. It's still an open question. There's a lot of papers. There's a lot of new ideas on this. I'm not saying that we don't know anything, but a lot of the classical tools aren't working in this era. And it's a great thing for us to try to make progress on.

So I'm going to use the final section to talk about some of the inductive biases and pressures that do lead deep nets to find solutions that generalize, that give us some inkling of where the theory might go and some practical tips. But the recap so far is that, empirically, deep nets generalize. They can make reasonable predictions, often accurate predictions on inputs that are not in their training data.

And you can even show this for large language models and things where they've been trained on so much data. But you can just make a counting argument that they still can't possibly be just memorizing. They have to be doing some generalization. And generalization, just from first principles, requires some kind of inductive bias. You have to have something that rules out the filing cabinet.

Only fitting the training data cannot fully explain generalization, because you can fit the training data by just memorizing the training points, but not having-- saying anything about the samples in between the out-of-distribution samples in between the training points-- or sorry, the out-of-sample samples in the test set.

So the inductive biases can't really just be about classical notions of model capacity, like number of parameters or VC dimension, which is number of functions that can be represented by that hypothesis space. So we need to understand what are the inductive biases, and how do they relate to meaningful controls on the expressivity and the capacity of the model? How do they prefer simple or smooth solutions over other solutions, and how do we measure that? And this is very much open and ongoing research. So deep nets must have some nice inductive biases. They must control the complexity in ways we don't fully know how to characterize. So what are these inductive biases that deep nets have that allow them to generalize?

So why do deep nets learn functions that generalize? And the basic question is, other than fitting the training data-- we understand that. That's approximation. We understand that fitting the training data is critical. That's part of the recipe for generalization, but there has to be something else.

Other than fitting the training data, what are the other pressures that affect the solution that deep nets arrive at? There must be some other pressures that affect which of the two solutions that both perfectly fit the training data does the deep net choose? And why does it choose the one that works and not the filing cabinet one, which potentially it would have the capacity to do in some cases?

So if we fit the training data, what's left to consider? So this purple space is meant to be-- it's the same space as I showed in purple dots before. But now, I'm going to give it this name, "version space." So the version space is the set of all hypotheses in the hypothesis space that fit the training data. So in neural nets, if I have an MLP, the version space is the set of all settings of the parameters, the weights and the biases, that perfectly interpolate, that perfectly fit my training data that achieves zero training error. So some blob within the set of functions I'm searching over, which is the set of weights and biases of the MLP, if that is my model.

So basically, the point is that some points in this set are bad and don't generalize, and some are good and do generalize. Why do deep nets find the ones that are good and do generalize to some degree? So these are all kind of ideas that are out there but are not yet completely standard or proven. But I think that they're fun to think about. So the first one is the idea that there's a simplicity bias in what's called the parameter function map.

The parameter function map is-- remember that we're trying to find functions that fit the data, but we're trying to find them by searching over parametrization of those functions. So we have access to the parameters-- the weights and biases. But every setting of the weights and biases defines actually some mathematical mapping f.

And there might be some parameter vectors-- two parameter vectors could define the same f. Like, if I have a network which just adds a bias of one on one layer and then subtracts the bias of one on the next layer, and otherwise it is an identity, well, that is equivalent to just a network that adds a bias of 0 and then adds a bias of 0.

So plus 1 minus 1 equals 0. There are a multiplicity-- this is a many-to-one mapping. There's a multiplicity of parameter vectors that all map to the same function. So the parameters and the function space are not the same thing. And we can look at the parameter function map.

And this is what was observed by these authors a few years ago empirically. They didn't prove this, but they observed it empirically. If you just sample a random set of parameters in an MLP or some other neural net models, most of them will map to functions f that are simple according to-- this is not number of parameters. This is not number of distinct-- this is not VC dimension. This is another one, Lempel-Ziv complexity.

It's like saying Lempel-Ziv is one of these compression algorithms. And I think this is saying like, if I run that function f on some data, how well can I compress it with this algorithm? So another crazy complexity measure that's hard to understand exactly how that would relate to generalization, but they tried that one. And so they're saying that-- this is like a probability distribution, I believe.

It's a little hard to interpret this plot, but there is high probability of finding-- if I take a random parameter vector, a random setting of the weights and biases in a network, most random settings will be down here. It'll be higher probability, and only a few will be functions-- will define functions that are really uncompressable and complex in that particular sense.

So here's a way to look at-- an intuitive way of looking at this. So if we randomly pick a point in the parameter space of the neural network, this will tend to map towards simple functions. So imagine that we're organizing this hypothesis space by the simple functions. The smooth compressible functions are down at the left, and more complex, wiggly, incompressible functions are up at the right.

So most random samples will be simple. And then what does learning do? Learning just moves our parameters and, therefore, moves the functions they map to toward the version space. It moves you toward the space that fits the data. And so we'll end up in this little section where we're at the corner of the version space where we have simple functions just by chance, because most random initializations of my parameters map to simple functions. And most of the volume, as I walk through parameter space, will also map to simple functions. And so just by chance, we will find a solution that is simple in that sense.

So I was involved in another paper that looked at that, but looked particularly at a measure of simplicity, which is the rank of the kernel. So I'm going to describe what that means. But we found that there's also this bias in the parameter function map toward solutions that are low rank in a certain sense.

So here's a MLP, W, ReLU, W. And here's a deeper MLP. And one of the classic stories-- And we'll come back to this in the representation learning lectures. But one of the classical stories is that shallow nets-- let's say I'm trying to classify between 10 different classes indicated by the colors. I'm trying to take my data and map them to a space where they're all linearly separable so I can classify them.

So the classic story is that deeper nets with more parameters have more capacity to organize the data and separate all those classes. So that's the picture here. We get better linear separation of all these classes at the top of the network. Now, I'm going to characterize that distribution at the top of the network with a kernel matrix, which is going to be the similarity between every data point and every other data point. So this creates a matrix, a bivariate function.

And it's called the kernel of my embeddings that the neural network arrives at. So every row and column is-- every row is a data point and the column is another data point. And the blue value is the similarity in the representational vectors at the output of that network. So we'll see these kinds of kernels a few times. This is a kernel that characterizes the representation at the output of the network over the input data. And if it's kind of block-structured like it is here, that means that it's found clusters.

So that's like all the red points have grouped together into one block, and all the purple points into another block. And that's good. So that's kind of what we're looking for if we want a good organization of my data that's easy to classify. It's very clustered. It will look block-structured like this.

So the common understanding is that deeper nets have greater capacity to organize the data into this clustered structure that's easy to then make a prediction on top of to classify. But the interesting thing is if you get rid of the nonlinearities, you actually get the same effect. So just a linear network-- just a stack of linear layers also will result in a deeper linear net-- results in more blocky kernels.

So we're going to characterize now the kernel with its rank. So the rank of that matrix is how many linearly independent rows and columns there are. So blockier matrices have lower rank. Lower rank is like, I have clustered the data into a simpler format. And that's useful for classification. So you get the same phenomenon with deep linear nets. And deep linear nets are equivalent to one linear transformation. So you can't explain this phenomena as the deeper nets have more capacity to organize the data.

So what's going on? The argument that we make in this paper is that products of matrices tend to be low rank. So again, it's like a parameter function map. If I have a random set of weight matrices, if I have more of them, just it will be higher probability that I will output vectors whose kernel is lower rank. So this is a property of random matrix theory that products of random Gaussian matrices will result in a low rank matrix.

Now, deep nets are not products of random matrices, but the full argument is that if I'm looking at the weight space of a neural network and I'm sampling a random point, if I have a shallow network, then most random points will be full rank. If I have a deeper network, whether it's a linear net or a nonlinear net, I will tend to map to lower rank solutions just by chance. Now, optimization will pick out which of the parameter vectors actually is in the version space-- actually fits the data.

But because most of the volume of parameter space is mapping to low rank solutions-- low rank representations of the data, when the network is deep, there'll be this kind of bias. There'll be the bias that if I'm just searching over some random location, I'll tend to be more likely to find low rank solutions than high rank solutions. And empirically, this is what happens.

So a lot of these things are only kind of provable and analyzable in linear models. But then we can show empirically in real deep nets. And what I'm showing here is the rank-- it's called effective rank. It's just kind of like the rank of the matrix, but a slightly different version of it. So higher rank means you have these representations of your data that are less clustered, and lower rank means this more clustered-like structure.

And on the y-axis is the probability. These are like probability densities. So this is empirical PDFs-- Probability Densities-- of if I randomly sample parameters of a neural network of different depths. So if I randomly sample the parameters of a depth-16 network, the functions that it maps to will be these functions that map to low rank embeddings. I know there's a lot of moving pieces here, but basically deeper nets will be biased toward functions that are simpler in the sense of having lower rank structure.

So one more way of looking at it-- so this is the rank which is in color and in height. If I'm navigating two different dimensions of my hypothesis space-- so my hypothesis space is parameterized by-- my parameter space has a million parameters, weights and biases, but I'm going to look at just two of them.

And I'm going to chart out as I walk around that space what is the effective rank. And the point here is that as I add depth-- so with a single layer model, I will have this kind of narrow ridge of low rank functions as I navigate my parameter space. But a two-layer model, I'll pull that ridge apart. And so more volume of the parameter space will map to low rank functions. And if I combine this with optimization, I'll just-- more of the solutions will lie in the low rank space as well.

Yeah, one set of ideas is that deep nets are biased toward simplicity according to Lempel-Ziv or according to the rank of the functions that they learn just because most parameterizations-- most settings of the parameters map to these simple functions. Another idea is that actually the optimization dynamics really play a big role here, and that I'm not just picking a random solution in my version space. I'm actually using an optimizer, which will prefer some solutions over other solutions.

And sometimes, I'll have explicit regularizers that prefer simple solutions in my optimizer or in my objective function. And so one of those is called weight decay. It's just saying that, as I optimize, I'll tend to shrink the weights a little bit on each update step. And this tends to make weights that aren't actually being used go to 0. And so I'll end up learning a set of weights of my neural network, which have lower norm and are simple in that sense. Often, we initialize our weights and biases of our networks near 0. And this acts like a bias toward finding solutions that have, again, low norm solutions-- low norm parameter vectors.

And then there's a lot of interesting work on the implicit biases of gradient descent and stochastic gradient descent. And so one idea here is that stochastic gradient descent-- well, gradient descent with a fixed step size kind of descend down. But let's say that I have a really, really sharp well in my objective function, in my energy landscape. Well, gradient descent can't go into a really sharp well if I'm using fixed step sizes. It'll just jump right over it. That well will be too small.

So what gradient descent with a fixed step size will do is it will tend to find these, what are called, flat minima. So kind of basins that are low loss, but are low loss over kind of a large region of the parameter space that I'm searching over. And these flat minima can be argued to generalize better than these narrow minima. The narrow minima might be more like just a weird solution that got lucky, but the flat minima is one that will generalize better.

So I'm not going to go into all the arguments for why that's the case, but you can see these papers if you're interested. But the optimizer is a critical part of the recipe too, and it can affect the generalization properties. But I think all of what I've said so far is not actually the most important. I think the most important is the following two slides-- is the architectural symmetries. And we saw these in ConvNets and graph nets, and we'll see these in transformers and other models too.

So our architectures are built to have certain invariances. Like, if I'm pooling over channels with a max pooling operation, I'll be invariant. And my filters are like edge detectors. I'll be invariant to the orientation of that bird's beak. I'll fire regardless of what the orientation of that bird's beak is. So max pooling achieves this invariance over the dimension we're maxing over.

Equivariances-- so graph nets and ConvNets have these equivariances. If I permute the labeling of the nodes, then I will permute the predictions. Translation equivariance-- if I have a filter, and I slide it across the image, then if I shift the image by some amount, I'll shift my predictions by that same amount.

So these are symmetries that are kind of built into our architectures that cause them to exhibit out-of-sample generalization, because I know if I just shift my image, it will respond in the way it was trained to respond on the unshifted image. That's what the symmetry gives you.

And then this kind of compositionality I was mentioning before-- that the ConvNet and a lot of architectures, they factor this input signal. They maybe chop it up into little components. They learn something about every little component, but then the conjunction of components is just given by something that's defined by the architecture. It's not learned. The conjunction of the outputs of these little patch-wise filters, patch-wise predictors is not learned. It's just you stitch them all back together.

So I think this is where really the real power and the real explanation of why things like ChatGPT, why they actually do these feats of generalization. It's not so much in my-- well, I don't know. But I think it's a lot about it factorizes the world. It carves the world at its joints. It factors its world into parcels, which are words, and sentences, and paragraphs. And then compositions of these things will be arrived at not from learning, but from just how the architecture is built.

And then a lot of neural networks have domain knowledge that's baked into them. For example, Sarah gave the example of the NeRF model, which is a neural architecture, but it uses equations of perspective projection, and it uses equations of light transport. And these are built into the process, and so it will generalize to new viewpoints because it's using structures. It's not just fitting to data. It's data plus structure, data plus constraints. And this is a drug discovery, drug prediction problem, where they've baked in structure. They know about how drugs are supposed to interact with each other. And the conjunction of fitting to data under structural constraints is what can allow you to generalize.

So I want to end with one final point, which is, well, maybe we can actually go back, ultimately, to this theory answer. The theory answer is that the shortest program that fits the data will be the one that generalizes. I said it's intractable, but what if we say, we don't have to find the shortest? Finding the shortest is intractable.

Maybe we don't have to optimize for the shortest program, but we can just say a short enough program that fits the data will generalize. And so this is something that I've just kind come back to over the years. This is a conversation I had with Ilya Sutskever, who's well-known as one of the founders of OpenAI.

And so he had this thing he said, which is deep-- this is not a direct quote. This is just the rough memory. "Deep nets generalize because they find small circuits that fit the data." And so at the time, I didn't quite get it. I was like, small circuits? What are you talking about? Deep nets are huge. Don't we need all of these inductive biases, and regularizers, and simplicity biases? Like, that seems critical to understanding deep learning. And we need to bake those in. We need to put those in our models.

And he said, "no, no, no, it's OK. Deep nets are finite. That's enough. Finite is small. Anything finite will look small if you have enough data." And I think that was a deep insight and maybe led to some of the success that he's had. So I'll end there. Thank you.