[SQUEAKING]

[RUSTLING]

[CLICKING]

PHILLIP ISOLA: Today we're going to have a lecture which is a little bit different than some of the other ones because it's a bit more on just practical advice and heuristics and hacks, a little less on formal knowledge and theory. So the big disclaimer is that this is going to be a somewhat opinionated lecture. I'll share things that I think are good ideas, but they're not what everybody would agree on. It's just anecdotes from my own experience.

And we'll talk about a bunch of different aspects of the deep learning pipeline and how to actually make working systems with data and models and good optimizers, and also then some comments on the other parts of the problem, like how do you create a good evaluation framework and how do you use compute effectively and so forth.

And these slides are put together from a lot of sources. So in particular, Evan Shelhamer had a talk which formed the skeleton of this talk that he was giving for some years. Evan was the developer of Caffe, which was the most popular deep learning framework 5 or 10 years ago. And then I got a lot of tips from Andrej Karpathy's blog posts on this topic and from my lab members and other folks.

OK. So the starting point of this lecture is that a big part of the story of deep learning is not just the theory or the algorithms, it's actually the hacks, the practical tips. It's a little bit of a triumph of the practitioners over the academics and the theorists. It doesn't mean that we won't eventually have a very clean mathematical theory of deep learning. But historically, a lot of the people that have really made progress here have just done these things which we might call hacking. And it's worth valuing that and understanding that.

And in the generalization lecture, I talked about this paper briefly, which is that you can overfit to a set of random labels with a deep net. And this means that the generalization bounds you get from some classical theories, like the VC dimension, are vacuous. They don't really tell you anything about how well that system will generalize. And yet, deep nets generalize. So clearly, the classical theory doesn't quite work.

And a lot of the places you may have been introduced to deep learning, if you've studied it in the past or you've programmed with neural nets in the past, would be courses that are very practical in nature, like fast.ai, which are really focused on, let's just code these systems and make them run, and maybe the math and the theory is secondary.

OK. So this practical stuff is a big part of the story, and it's worth understanding. So I'm going to start with one story, which comes from my postdoc advisor, Alyosha Efros, here on the right, back when I was at Berkeley. And his advisor, Jitendra Malik, on the left, was mentoring Alyosha when Alyosha was a grad student. So this has been passed down to me through multiple generations.

And essentially, Alyosha was training models back in those days, 10, 20 years ago and would come to Jitendra, his advisor, and say, OK, here is my result. And it wasn't a deep net. It was something else. But this is just meant to convey the idea that he would show the results of some numbers, some plot, like, oh, I got 30% accuracy. Or maybe here's my loss or my optimization graph. And Jitendra would say, OK, what's going on here? Why are these spikes? And Alyosha would say, well, I don't know.

And so Jitendra said, OK, this is not enough. This doesn't tell you that much about what's going on. You really have to look at the data. You have to look at the results that you're getting. This was a computer vision lab. And so the data was images. Look at the images. Look at what images you're analyzing, what labels you're predicting. Look at the data. Don't just give me some summary statistic. And he put it this way. He said, "Become friends with every pixel."

So I think that's just a lovely phrase. And whatever domain you're in, try to become friends with the data points in that domain. If you're studying music, become friends with every note. If you're studying chemistry, become friends with every molecule. Become friends with every pixel. Look at the data. OK. So then Alyosha is well known in computer vision now as being a very intuitive and insightful person, in terms of the holistic nature of computer vision. He went to France and took photos and understood the beauty of that data type.

OK. So look at the data. And you can think about this also for your final projects. Don't just show us a plot. Show us the data, something a little bit more high dimensional than just a single plot. Here's an example from a system that tried to diagnose whether or not a scan of a patient's chest-- this patient has breast cancer or not. And so you're trying to classify benign or malignant for the possibility of there being a tumor.

And this paper is actually an analysis and critique of prior work. So it's not their fault. These authors were the ones that pointed out this issue. But what happened is, a deep net was used in prior work to try to identify, is this cancerous or not? And it got really high accuracy-- I don't know-- 99% accuracy. So what do you think it was doing. Anyone maybe have seen cases like this? Was this deep net accurately classifying the scan? Actually, I'll tell you. I'll tell you, it wasn't. So they then tested this system, and it failed.

So why did it fail? What was it doing that made it fail? I saw an answer back here.

AUDIENCE: So [INAUDIBLE] nothing is cancerous. [INAUDIBLE]

PHILLIP ISOLA: Yeah. So one answer-- it just said nothing is cancerous. And maybe the data set is imbalanced, so 99% of the cases were not cancerous. That could have been it. That's not the case in this one, but that is a common failure. Let's go back here.

AUDIENCE: It was looking at stuff outside of the lungs. Like, maybe the R in the top right corner is more often seen in cancers.

PHILLIP ISOLA: That was the point they were trying to make. So in this case, as you said, the deep net was not looking at the tissue. It was looking at the R in the corner. So there was this kind of spurious correlation, this shortcut. And it turned out that that R was some indicator about what hospital this came from or-- I don't remember entirely. But let's just imagine that it was R if it came from Roosevelt Hospital or something like that. And at Roosevelt Hospital, all the patients that were put into the system had malignant tumors because it was a hospital you go to for that situation.

So there was a bias in the data, and the data gave away the answer. And so you really need to look at these things to identify, is your model working for the right reason or the wrong reason? This is the wrong reason because it won't generalize to other hospitals. OK. Or you can imagine a worst case, where R just meant malignant, and it would have a different symbol if it weren't. It would just completely give away the answer, and it wouldn't generalize to a system that doesn't give you the answer. And that's what you really care about.

OK. Look at the output too. So don't just look at your loss curve. That's the most superficial thing. You should definitely do that, but don't only do that. So here is something that would be better to look at if you're in the business of training generative models of cats. We'll talk about generative models later. But these are models that make photos of-- you've all seen this, DALL-E and Midjourney and these types of things.

So this is learning to make photos of cats. And if I were to only look at my loss, like, what does my objective function say about the quality of these images? It might be spiky and crazy. I don't know what's going on. But if I look at samples from the model-- actually, these are outputs from the model as it's being trained-- you get a lot more information. You can kind of see something, which is like, it seems to oscillate. It's, like, periodic.

So now this happens to be what's called a generative adversarial network. We'll talk a little bit about these later. But they do have this oscillatory behavior. And it's something that's very obvious in this output. It would also maybe be obvious in certain loss curves. But you get a much higher-dimensional, high-throughput understanding of what's going on when you actually look at the data like this.

OK. Oh, here's another example. These are some little ants that I trained years ago to try to play some predator-prey game. And what was happening is they would-- I looked at the reward function. This is like a reinforcement learning. They're trying to chase each other. I think that the blue ones are trying to run away from the red ones, which are the predators.

And at first, the reward of the predators would go up a little bit, but then it would just plateau. And I didn't know what was going on. And why was it plateauing? Because they were falling over and then making no more progress. So if I only looked at the loss, it was like, oh, it just plateaued. Learning is hard. I don't know. SGD was failing. No, it was failing for a very particular reason, which is that the ants fell over and I had to fix that.

OK. So look at your outputs. Look at your inputs. Visualize what you're doing. OK. So just to summarize some of those points-- look at the data. Inspect the distribution of inputs in your training set. Inspect the distribution of outputs in your training set.

As the student said over here, you could have an imbalanced set of target outputs in your training data, and that would be another common failure. If your data is 99% benign tumors and 1% malignant, then chance, where the system is completely doing naive guessing based on those priors, not looking at the data, not looking at the input at all would be 99%. You could just say, well, I know I'll be 99% right if I always say benign.

Then you have to calibrate with respect to that. You have to realize that 99% is trivial. And therefore, you're looking for a higher number than this. Or maybe you want to measure something else, like the conditional probability of being correct, given that it's malignant.

OK. So you can select random mini batches. You can visualize them, plot histograms, plot marginal statistics. What is the variability of the units of measure in your data? Are your numbers in the range 0 to 1, or in the range 0 to 256, or in some other range? Can negative numbers be your data, or is it only positive numbers? These are the types of things you want to know.

OK. So one of the most common bugs that I come across, especially in final projects, is that the data is not being preprocessed and loaded into the format that you think it is.

So, for example, let's say you're working with images, and you have assumed that your data lies in the range 0 to 1. But actually, when you loaded it, it is loaded as a unsigned integer 8, uint8's in the range 0 to 255, which is a common format for imagery. So you might have a bug because you have code that-- let's say you actually clamp your inputs to be between 0 and 1, but the data you're loading is between 0 and 255. So now you're losing all of the information because you're making all your data points equal to either 0 or 1.

So you have to be very careful about this. And what I would recommend-- and I think it's just a very simple recipe-- is, somewhere in your code, you're going to have model.forward, right? You're going to run your neural net in a forward pass on your data point. Right before that line is where you should inspect the data. So you don't want to inspect the data as it's sorting your hard drive. You don't want to inspect the data as it's initially loaded by your data loader.

You can do that too, but the place where you really need to understand what the format of the data is and what the distribution of the data is right before you run your model on it. And make sure that what format you think it is, what shape you think it has, what dimensionality you think it has, exactly matches what it actually is in practice.

OK. So here's just another example of where this can go wrong. So it turns out that in some of these old neural net libraries-- one is called DeCAF. Another is called Caffe. So these are like the precursors of PyTorch and TensorFlow and other models like that, other systems like that.

If you load an image off of your desk, in DeCAF, it will be represented upside down. It'll be flipped. And in Caffe, the channels will be permuted. So the standard color space for Caffe is BGR instead of RGB. The color channels are permuted. And so if I have an image on disk and it's in RGB order and then I load it into Caffe, the order of the channels will be-- the blue channel is in the first channel dimension, the first index.

So that was because of some old convention in a library called OpenCV. It doesn't really matter, but it used to be BGR was the convention, and now we use RGB as a convention. But let's say that I'm training my Caffe model on images. And maybe it says-- oh, this is a blueberry, right?

Well, that could be because it's actually seen an object which has a strong red channel, but it's interpreting that red channel as the color blue because maybe the system was pre-trained on data in the format RGB. But now you load it into Caffe, and it's in the framework BGR. So anyway, you can get errors like this. So you really need to have a match between what your model is expecting and what you're giving it. And if there's any mismatch between training and inference, or between the model architecture and the numerical transformations you apply and the data format, then this will cause problems.

OK. So here's, I think, the most important function in deep learning. This is one that I wrote. So it's a very clever advanced contribution. OK. So here it is. Inspect data. OK. So I just put this all over in code that I write. Right before you called model.forward, that's the most important place to put this. But you can just sprinkle it all throughout your code.

OK. It's going to just tell you a few things. And to me, these are the most important things to know about your data, but there's a lot more you could think of as well. Again, you could try to visualize it somehow or plot two-dimensional distributions over the different axes. But, OK, what's the data type? Is it a tensor? Is it a 32-bit tensor? Is it a uint8? What is it? What's the shape? The shape of the tensors in deep learning are super critical.

Does it require gradients? So in your problem sets, you're constantly training these networks. And any parameter that you want to train needs to have requires gradient equals true so that you're actually getting gradients for that parameter. What is the min? What is the max? What is the mean? What is the variance? You could go on from here, but these are the ones I like. There are some libraries that just kind of give you this by default. But I find those sometimes change or have to require an install, or it's just a little bit-- I just like the snippet of code. That snippet of code will work, I think, for at least a few years. OK, so really simple, but I actually think this kind of stuff is important.

OK, so data. Another thing is, again, the numerical range matters a lot. So one of the standard and most useful ways of removing dependence on exactly the units in which you're measuring your inputs and your data in general is to normalize it somehow. This is like the standardization transformation, where I take my data and I subtract the mean and divide by the square root of the variance. This is a really common first step to any kind of data loading that you're going to do. You load your data. You subtract the mean, divide by the variance.

And the point that this achieves is that it makes it so that you're kind of invariant to exactly the unit of measure in which the data is stored. So if I'm modeling some geospatial information, it doesn't matter if I've measured my data with meters or centimeters or inches or feet. If you standardize, then it will remove the dependence on the scale of those measurements to some degree. At least, it, makes it unit variance and zero mean. You could also squash and squish in other ways to massage your data into a more canonical distribution. Yeah, question?

AUDIENCE: I was going to ask, is this necessarily better than just a min, maximum--

PHILLIP ISOLA: Yeah, exactly. So is this better than just divide by the difference between the min and the max or subtract the min divided by the difference between min and max? Not necessarily. Basically, you want to remove information about maybe the unit of measure, like the scale of it, or the mean of it, or possibly other statistics of it. And there's a lot of ways of doing that.

So I'd say this is one of the most common, is just standardization. And the other most common is probably subtract the min so that the numbers are all positive now and then divide by the max after subtracting the min so now the highest value is 1, and the lowest value is 0. That's another very nice trick. Now your data lies in the range 0 to 1.

OK. And this kind of removes inductive bias about the unit of measurement, which might be good or might be bad. So it says that no one dimension is treated as more important a priori than another dimension. They're all in on the same scale. They all range from 0 to 1. Can anybody think of a case where this would be bad, where, actually, you would want to use that information about the scale? OK, yeah?

AUDIENCE: So if you have, say, like a horizontal and vertical distance, if they are actually two-dimensional phenomenons, you wouldn't want to scale them differently.

PHILLIP ISOLA: Yeah. If you have horizontal and vertical distance, you might not want to scale these differently because maybe distance in the vertical dimension is physically different in meaning than distance in the horizontal dimension. Is that what you're getting at?

AUDIENCE: If they're physically the same, you'd want to scale them together as opposed to independently.

PHILLIP ISOLA: Oh. Yeah. So if they're physically the same, then it could be that the horizontal data has a very narrow distribution. But you wouldn't want to therefore scale it up because now it's like you're measuring space in the horizontal axis differently than the vertical axis. Yeah. So there's a lot of interesting considerations that can come up where this could go wrong.

But I would say, in general, it's a good heuristic, especially when you have measurements on vastly different scales. Like, you have some measurements that are 10 million times bigger than other measurements. It'll be very hard for your network to learn to pay attention to the tiny measurements if there's another measurement that's 10 million times bigger, because the gradients will just be much, much smaller for the data on a smaller scale.

OK. This slide is going to be a little more subtle. So beware of low dimensions. This is a slightly different thing than maybe what you've heard before. Maybe you've taken some statistics class or even high-dimensional statistics and people said, oh, high dimensions are weird. Beware of high dimensions. Things are different than your intuitions in high dimensions.

OK, that's true. So the human intuitions we have are mostly for low-dimensional, three-dimensional spaces. The world is, like, three spatial coordinates. Time is another dimension. We're used to thinking in low-dimensional spaces. We think of objects like a sphere in a low-dimensional space. We know what that looks like. But you may know from other classes that a sphere in high-dimensional space has slightly different properties.

So a Gaussian distribution in high-dimensional space is like a soap bubble, where almost all of the probability is in a tiny little band around the surface of the sphere. OK. So things are a little bit different in high-dimensional space than we're used to. But what I want to point out is that if we weren't human and we're just some ideal creatures of math, then high dimensions is a simple one. Things are simple in high dimensions, and they're weird in low dimensions. And deep learning is all about getting used to thinking in high dimensions. And everything gets simple in high dimensions. So that's what's really nice about it.

So here's an example of that. So we talked about RMS-norm. Remember, Jeremy introduced that in one of his lectures. And we sometimes will want to take the RMS norm of the activations to achieve good effects on learning dynamics. And we also talked about layernorm, which is the standard layer in transformers, which is almost the same. And in high dimensions, these things behave almost identically. So the only difference is, layernorm, you subtract the mean over your vector, and then you take the RMS-norm of that vector. In RMS-norm, you don't subtract the mean of the vector.

So RMS-norm will tend to map data points onto the unit hypersphere. That's the effect of that operation. So here's a visualization of the inputs. It's like a cloud of data points. And then the outputs, after an RMS-norm layer, are going to map everything to this unit sphere. So that is this normalization. You get everything onto the unit.

Well, layernorm, you might think, OK, yeah, it's basically the same operation. And it is in high dimensions. But in low dimensions, it has this effect. So for 2D inputs-- this is a layernorm applied to 2D inputs. The data points are the red points. But layernorm is doing is by subtracting the mean-- like, the vector x is two-dimensional for a 2D space. And we're subtracting the mean of a two-dimensional vector. I'm losing one degree of freedom. And now that kind of degenerates into all of my outputs have to be on-- I lost one degree of freedom, essentially.

And then the RMS norm kind of loses another degree of freedom. So now the outputs have 0 degrees of freedom, and they end up being on these just two points as opposed to being on a one-dimensional manifold, the circle for RMS-norm. OK. So the point is just be careful in low dimensions. Try to work in high dimensions as much as possible, where things are nice and simple. And get an intuition for high dimensions. And especially, normalization layers, like batch norm, won't work well if your batch size is small. layernorm won't work well if your layer dimensionality is small. Or it will have weird effects, like this. So you have to be really careful.

I remember I had a bug-- so the work that was the most popular that I've ever been involved in was this project called Pix2Pix. And in the initial code release of that project, I had batch norm over a batch size of 1 for one of the baselines. And batch norm over batch size of 1 means take a data point and subtract the mean over a batch of 1. It just take the data, subtract it. It's 0. So that baseline was trivially beaten. And I corrected it. We still beat the baseline in the end. But that was out there for a while. And that was just a bug. So you can't take batch norm over batch size 1. You can't do layernorm over layer size 1 if the dimensionality layer is 1.

OK. So the general trick is just you're going to be working with data tensors of batch size N by M by C, maybe some other dimensions as well, these tensors. The weights are going to be matrices N by M. Just make sure all of your dimensions are big, and you'll be in good shape. Yeah?

AUDIENCE: When you say big, what sort of scale do you mean?

PHILLIP ISOLA: Oh, yeah. When I say big, what kind of scale do I mean? Yeah, these are just heuristics. So I don't know. Let's say at least 10 and above, and ideally, probably as big as-- the bigger, the better, in general, I would say. Asymptotically, everything becomes nice and simple. Deep learning is kind of like thermodynamics. It's really hard to say anything about a set of three or four molecules. The three-body problem in physics is really hard to solve. But it's very easy to say what the temperature is of a billion atoms because, asymptotically, things become very simple. We leverage the law of large numbers. So it's the same thing. You always want to be toward the asymptotic limit.

OK. What is the cost? Well, money, time, energy. So there is a trade-off. There's not zero cost to increasing the scale of these things. But for statistics and inference and prediction, bigger is better. Yeah. These are basically things that I've already said. But you want to know the range of your data. You want to know the shape of your data.

And one little trick-- this is coming from Evan-- is that if you want to sanity check your neural net architecture, don't test it on tensors in which different dimensions of the tensor all are the same number, because it makes it much harder to catch errors with reshaping.

Because it could be that you do some reshaping operations and you still get a tensor of dimension 64 in the second dimension of that tensor. But the first dimension also should be 64. And you actually have permuted the tensor in the wrong way, but you can't tell. So you get kind of a better type checking to make sure the numbers all work if you use a unique set of numbers, a unique dimensionality for each dimension of the tensor.

OK. So it's a good way of just getting syntax errors in your code. And they'll throw a shape mismatch error when you've used unique numbers if you're getting them in the wrong order. But they won't if you have the same dimensionality. OK. And yeah, be very careful with changing the types of your tensors. If you cast a tensor as a uint8 because you think I'll save memory, well, uint8's can't represent negative numbers. So now all your negative data, if you have negative data, will go away.

So that's the point here. What is negative 1 for a byte? A byte is like a uint8. And what if I've taken my input data-- I'm like, oh, I know my input data is positive, but now I standardized it to make it lie between the unit variance and centered on 0. So that means it can have negative values. But I still have the intuition, well, I'm working with only positive data. But I've standardized it. Well, now it has negative values, but it's uint8's, and it can't have negative values. OK, there's going to be a bug.

OK. So you've probably gotten the feeling for this in the Psets already, but a lot of your code is just going to be reshape operations. And this becomes a bookkeeping nightmare. You're going to have a lot of bugs that come up just because you didn't reshape things in the right way. You're constantly transposing, permuting, reshaping, flattening, unflattening, unsqueezing, squeezing. This is like half the code in PyTorch.

And I personally can never remember the conventions for these things. So if I do reshape a tensor-- let's say it's a two-dimensional tensor. And I reshape it into a flattened array-- so just a vector of the product of the two dimensions-- is this going to stack the rows into a sequence or stack the columns into a sequence? I don't even remember. I don't know what this does. Who knows, actually.

Is it going to be row major order or column major order? If I have a matrix, is it going to read off the first row and then the next row and put it after the first row? Does anyone know for reshape in PyTorch? Yeah?

AUDIENCE: It's row contiguous.

PHILLIP ISOLA: OK, it's row contiguous So it does the first row, and then it takes the next one and puts it after it. Right. But it's hard to remember these things. And you really don't want to have to remember all that stuff. So this is a library that I think makes some of that a lot easier, makes keeping track of the reshape operations a little bit easier, is einops. It doesn't mean you don't have to think at all. And you can still have bugs with einops. But I found it to be very helpful.

So all of the kind of reshape, flatten, unsqueeze operations now have a common notation. They're just rearrange. And you give a string, which is going to say-- it's kind of like a shorthand for telling the system what is the set of reshapes that will map from one space to another space.

So here, we have b, h, w, c, That means that, first, I have a four-dimensional tensor. And I'm going to map that to h, b, w, c. b, w are in parentheses. That means I'll flatten those two into a single dimension.

So it'll be an h by b times w by c tensor. It'll be a tensor with three dimensions. And yeah, you still have to worry a little bit. Is it the row major order or column major order? But it makes a lot of this easier to think about. And it tends to have the property that-- or maybe it always has the property that if I do the inverse operation, it will be an identity.

So if I do reshape b h w c to h b w c and then I do the inverse, where I do h parentheses b w c and then to b h w c, so I do the opposite direction-- if I do those one after the other, I'll get back where I started. They're inverses of each other. So that's a nice property. OK. Reshape might have that property too, but this makes it very easy. OK. So try einops. OK. As you can see, this lecture is just going to be throwing a whole lot of little tips and tricks to you. And I'm hoping that some of them will stick and you can go and pull on these later.

OK, so data augmentation. You did some data augmentation in Pset 1, so you know what data augmentation is. Just a quick recap-- if I want to take small data and I want to make it bigger, there's simple tricks which you can do, which are apply transformations to your data, which should not fundamentally change it. So you want to be able to apply transformations to, in this case, images that do not change the target class. So this creates bigger data from smaller data.

And I think a lot of people have this impression that data augmentation is a little bit ugly, a little hacky. And there's a whole community of people who say, oh, you should really replace data augmentation with what we might call geometric deep learning, where data augmentation is forcing invariance to this set of transformations, forcing invariance to mirror flips and crops and changing the lighting conditions very slightly, like just changing the intensity.

So data augmentation says all of those changes should have no effect on the label. But geometric deep learning approaches say, no, I should design an invariant neural network architecture. I should use a ConvNet to achieve translation equivariance and then pooling into two translation invariance. And therefore, why do I need this cropping? And there's all these arguments that I can actually use an architecture to enforce the invariances and equivariances and symmetries that I care about as opposed to data augmentation.

But I think that that's mostly not the right advice for a good hacker in machine learning. I think the right advice is, data augmentation is really easy, really intuitive. And it's architecture agnostic. That's the power of it. It's all on the data side. So you don't have to design some fancy symmetry-preserving architecture graphnet that knows how molecule chirality should be properly propagated. You don't have to do any of that. You just say, OK, here's the setup of things I think are fine to do to my data. And I'll feed that to any architecture. I can send this to an MLP, and it will be invariant to these operations too.

So data augmentation, I think, is the way to go for most of the types of invariances you actually want. It's just simpler. And software engineering wise, it decouples these properties from the architecture design. OK. That might change. Maybe in a few years, we'll have a very clean framework for geometric deep learning, and then we'll go to that.

OK. But I love data augmentation. I think it's one of the most important tools. And there's another kind of data augmentation which is really cool, which is sometimes called domain randomization. Oh. OK, yeah. So I'm just showing-- here's a visualization of what data augmentation is doing. But I think that you probably have thought about this before. But we have our training data. I just make more data from little data by adding these other points by perturbing my input training data. And that just makes it so that my test data looks like it's in distribution. My test data just looks like more training data.

OK, so a little quiz now. So which of these three loss curves would you be happy with? Let's say you're running an experiment. You're training your model. Which one are you going to be happiest with? So raise your hand if you think the leftmost image is the one that you're most happy with. OK. We get, like, a third. OK. The middle one, are you happy with? OK. No one thinks the middle one. The right one? We get 2/3.

OK. So you're all actually very clever. The first guess, the naive guess, would be that the left one. And that is the right answer in some sense, but it's not the answer I was looking for. So what is wrong with the left curve? OK. So it's too easy. If you're seeing a loss curve that just goes down really fast and then it's just flat, well, you've wasted time optimizing in the flat region. I mean, maybe not. Maybe it's decreasing a little. But if it's truly flat, you're just wasting flops by doing more training in that region.

And in general, you don't want to make your problem so easy that training on it converges really, really quickly. You want to make your problem hard enough that you're actually going to learn a lot. So, OK, obviously the middle plot is bad because your loss is going up as you train it. But the kind of plot you want to see is that, yes, your loss is going down. It's very graceful and smooth. But you're making continual progress, and there's a long way to go. And you've made the problem hard enough that there's something interesting to learn.

So here's a rough recipe I have in mind. So you definitely want to minimize your loss with respect to your parameters. That's backpropagation. That's basically the main optimization thing we do in deep learning. But there's another thing that we don't usually optimize. But if you're going to actually work on a largescale project, most of your time will be on this part, is changing your data to make it harder. Make your data hard enough that there's a lot of information you can get out of it.

So you don't necessarily want to max over data, but you want to increase the difficulty of your data until there's something interesting to learn from it. And that's what domain randomization, that's what data augmentation do. They add more data so the learning problem becomes harder. They're a little bit like going from the really bad, easy loss function on the left, loss curve on the left, to this one over here. Why? Because you're adding more data points that the model has to learn about. And why is that a good thing? Because it will generalize better. Right?

So fitting this distribution, understanding that is easier than this, is easier than this. But the last one, which is the hardest to train on, is going to generalize the best. OK. So another version of that idea goes under the name domain randomization. So this is a popular term in robotics, where we want to train a robot to be able to pick up blocks on a table. And I could train it with just one lighting condition with just-- this is training in simulation and testing in reality. I could train it with just one particular setting of colors of the blocks. But then it will only know how to pick up blocks of that color. I could train it with one lighting condition, but then it will only work when I'm in that lighting condition.

So I make my problem harder by adding all this random variation. And it will take a lot longer to train, but it will generalize better to any new test environment with different color blocks than it saw at training time. Yeah?

AUDIENCE: Will it generalize even to lighting solutions that will never happen?

PHILLIP ISOLA: Will it generalize to light changes that never happen? So the idea is that you want to make it so that you've covered all possible lightings in your training data so that the test data just looks like another random lighting condition that exists in the training data. So the short answer is, no, it won't generalize to a lighting configuration that's not seen in this randomized training data.

But the longer answer is, you potentially could get the property that if you train on a broad enough distribution, maybe you start to get this kind of emergent generalization. I think that I'll set that aside, but I'll say the short answer is no. But maybe there are cases where you would actually generalize even better out of distribution with respect to this randomized training distribution. But it's a little more subtle, and I'm not sure that there's a clear answer for that.

Yeah. OK. So what is this doing? We're just minimizing the gap between the source domain, which is simulation, and the target domain, which is reality, by randomizing the source domain. This is what OpenAI used for their robot hand some years ago. And I'll just show you what their data augmentation looked like. So they just had a simulated environment. They rendered that hand with a lot of random conditions. They also randomized the physics parameters. So they randomized gravity.

So why would they randomize gravity? We know that gravity's negative 9.8 meters per second, whatever it is. And why would they randomize gravity? What do you think? Let's go here.

AUDIENCE: Space?

PHILLIP ISOLA: Oh, right. OK.

AUDIENCE: [INAUDIBLE]

PHILLIP ISOLA: So the answer-- you go to space, change the altitude. Maybe you're on a different planet. That's one good answer. So like, if they want their robot to work on Mars, this will work better. That's not quite, I think, what they had in mind. They weren't quite imagining going to Mars yet. But-- OK.

AUDIENCE: When a machine is moving, are you-- [INAUDIBLE] they can have some acceleration, can change the gravity.

PHILLIP ISOLA: That's a really good point. So when the machine is moving, then your measurement of gravity is affected by that because of some general relativity type-- like, reference frame. You can't tell the difference between acceleration and gravity, according to physics. So that's another possibility. I think that's part of it.

But what I think is the real reason is gravity-- they have these accelerometers that measure things, like how the robot's moving and what the gravity direction is, what's down, what's up. And it was just a little bit noisy. So they're just modeling that noise, essentially. They're saying, well, actually, our physical measurement of gravity might be pointing in the wrong direction. Maybe 9.8 is always correct, but let's say it's slightly miscalibrated, so slightly in the wrong direction. Well, if I trained so that it's always slightly randomized, it doesn't matter if I'm miscalibrated. It'll still be within distribution, and it'll still be robust to that variation.

OK, a lot of good answers there. But here's the one that I thought was the most important kind of surprise or the most important lesson, is that-- this is on the x-axis, the years of training, simulated years. So it trains in a few weeks, but it was simulated years of experience. And on the y-axis is the performance. So higher is better. And with no randomization, it learns really, really quickly. So this is like that loss curve that goes down really, really quickly.

And some engineer on this project might have seen that curve and been like, OK, we're done. We solved it. But they were smarter than that. And they said, no, no, let's add a lot of randomization and make it robust to all the different errors in gravity that we might be measuring and make it robust to all the lighting conditions. And then-- look at this. It took, like, 10 times longer to get to the same level. Like, this was one year of experience to get to there, let's say, and then 10 years. This is a logarithmic scale.

So they're having to train this thing for so much longer to make it solve a much harder problem, which is to solve robotic cube manipulation under a lot of different conditions as opposed to just one condition. And if you reported that to your manager and you're like, oh, I got a lower accuracy, well, a bad manager would say, oh, get rid of that. Make it learn quickly. But that's wrong because you want to make the problem hard enough that it will actually generalize to the real cases you care about, which are the test cases, not the training cases. This is performance on the training data. OK. So high training accuracy means the problem is too easy. So make your problem hard enough that it will generalize better.

OK. So let's see, continuing on with this theme. So in academia, in universities, in research labs, most of the time, in machine learning and deep learning, we do the following. We say we have fixed data. You're given data and you have to model it. And you design neural net architectures and learning problems and objective functions and optimizers to learn a function that models that data.

But this is actually not the way it works in most real-world scenarios in industry in practical scenarios. It's usually the other way around. You just download the latest, greatest model, and you don't change it too much. But what you do change is the data. That's what companies can do. That's what governments can do. We can change the data. And that's where most of the leverage, in my opinion, is.

So you can collect more data. You can collect better data. You can label your data in new ways. You can change the data. And this is an even more powerful set of tools than changing the learning algorithm. So most of this class and most classes you'll take at MIT are about the algorithms and not the data. But there's, I think, more leverage to be had on the data side in practice. OK. So it's probably a problem in our curriculum, and we should have more data science or data-centered classes. And we have some. We'll talk a lot about data properties and things like this too in this class.

OK. So here's another little quiz. So here is a language model that's trying to predict the next word in these sentences. And these are the first line of famous novels. So what do you think is the second word after it? Let's say I'm going to try to predict next word, given previous word. What do you think it's going to be? This is a famous opening to a book.

AUDIENCE: All of them.

AUDIENCE: [INAUDIBLE]

PHILLIP ISOLA: It was. Who knows what the book is?

AUDIENCE: *Tale of Two Cities.*

PHILLIP ISOLA: The--

AUDIENCE: *Tale of Two Cities.*

PHILLIP ISOLA: *Tale of Two*-- wow. OK.

[LAUGHTER]

Yes, that's right. You're right. But there's actually a lot of other ones. There's a lot of first lines-- so anyway, yeah. "It was the best of times. It was the worst of times." That's actually what I had in mind. But obviously, it's not the only answer. What about the next one? "Call me"--

AUDIENCE: Ishmael.

PHILLIP ISOLA: Ishmael. OK. That's *Moby Dick.* "All happy families"--

AUDIENCE: [INAUDIBLE]

PHILLIP ISOLA: OK. And hopefully you can hear from the volume that a lot of people know the last one. Not as many knew the first one. What's the point? The point is that prediction is easier the longer the input, OK. Because the longer the input, the more information you have with which to make that prediction. And this is also something that people got, I think, a little bit wrong historically. It was kind of thought that, well, it'll be harder to model strings of length n than strings of length m less than n.

So back in the early days of NLP, people would have these N-gram models. And it was like a model of how words co-occur with each other. And you'd have bigrams. How often does this word co-occur with that word? And trigrams and N-grams. And you didn't want to make N too large. But this was a little bit misguided, I think. Really, you can make your prediction problem much easier by making N really, really, really big.

There is a little technical distinction, which is, in the N-gram error, you are modeling the joint distribution of everything. Here, you're modeling just the conditional distribution on the last word given the previous words. But I think it threw off our intuitions a bit that the more data you put on the input, the easier the prediction problem becomes. So you should add a lot of data on your inputs.

And you can do this. In research, maybe you just have a fixed standardized data set. But in life, you don't. You can change your data. And that's the main thing that's going to give you performance. So we have X, and we have Y. We're predicting Y from X. So suppose you're designing a pharmaceutical drug. And Y is going to be the effectiveness of the drug.

OK. You generally can't change Y too much. Like, maybe your boss just assigned you that job. You don't get to have too much control over what is the goal that's useful to society. But you can change X a lot. So what would you put in X? Let's say I want to predict the effectiveness of a pharmaceutical. So maybe I'll put the chemical formula. What else might you put in to make it an-- like, if I only do that, OK, I can kind of predict the effectiveness. What else might you put into X to improve your predictive performance? So somebody shout out some ideas.

AUDIENCE: Patient data.

PHILLIP ISOLA: Patient data, yes. Patient age, patient-- you can take a biopsy, personalized medicine. You're going to get better predictions with more information. OK. What else? The folding structure of the protein, so not just the chemical structure, but the geometry. What else might you put in?

AUDIENCE: Other medicine that worked.

PHILLIP ISOLA: Other medicines that worked, nearest neighbor, similar medicines. Yeah. OK, everything, the universe. You should put the universe in, OK? So the point is, the more information you condition on, the easier the prediction problem becomes. Now, this contradicts my previous advice that you don't want to make your prediction problem too easy because you might overfit to shortcuts. Like, you might be putting in the R label at the top of the cancer scan. But I think, roughly, this is even better advice, that you should put a lot of information into the inputs to your modeling problem and control overfitting in other ways.

OK. So why is this a good recipe? So if I'm going to map from X to Y and I don't have a lot of information about X, that means that there could be a lot of possible predictions that are all equally valid. So if I just know the chemical formula and I'm trying to predict if it's a beneficial chemical and I don't know anything about the patient, then I have to say, well, with 10% probability, it will help. With 90%, it won't help. And when it doesn't help, it might not help in these different ways. There's a complicated amount of uncertainty in the output space.

And with neural nets, you can always think of it or you can often think of it as modeling a probability of Y given X. We'll talk more about this in the generative modeling lectures. But if my probability, my uncertainty that I'm just trying to model is complicated distribution, then it's just very hard to solve this problem.

But there's a hack, which is that standard neural net training that we've learned about so far will use things like L2 loss function to do regression, take some inputs and make a point prediction of the output value. So it doesn't handle modeling a whole distribution of possibilities. The standard tools we have so far only do this. We'll learn about generative models that handle uncertainty later, but that will be more complicated.

The simple models do this. But you can solve this problem with the simple models. And the way you do it is, you just put enough information in x that probability of y given x becomes deterministic. It becomes a delta function. It looks like a single point. And this hack works really well. This is the hack that led to text-to-image models working so well.

So here's the history of text-to-image generation-- or image generation, let's just say. So in 2019, the best image generation, it looked pretty cool at the time. But in retrospect, it's just not much. All it could do is make photos of faces, front-facing photos of faces, good lighting. It could just make this one really narrow kind of thing in the world. Imagine all the things you might want to visualize. It can only make faces.

And what they were doing at that time was they were training models that don't map X to Y, that just Model Y, just model the output. So the faces are the output. So we had no information about X, and we just have to model the entire possible space of all faces. So you can't do that for all images in the universe. You can only do that for really narrow categories, like all faces, with current technology.

But now we do have models that can make all kind of crazy images, any image you can imagine, right? DALL-E, Stable Diffusion, Midjourney. So what happened, what allowed that to happen is because we switched to giving a lot of information into the input to make it so that the problem of modeling data is almost deterministic. There's very little uncertainty about-- well, not very little. There's still some. But there's not as much uncertainty about how something should look if you have a sentence describing how it should look. That makes it so that standard neural net methods actually work.

OK. So I don't think we've made that much progress in generative modeling on modeling complicated distributions. We've actually just switched to modeling simple Gaussian distributions conditioned on a lot of information. But we'll talk more about this when we get to generative modeling. The point is that you can change your data to make things work better. And the big change that I'm recommending is, make it big. Big means train on a lot of data. That's the most obvious sense of big. But it also means that your inputs can be a big object, a high-dimensional vector. Remember, high dimensions makes things easier-- a lot of information in the input to pin down the output so there's not uncertainty in that mapping.

And then this one's a little bit more subtle, but I also would recommend making Y a big object. But we'll get to that when we get to generative modeling and foundation models and things like this. Yeah, question?

AUDIENCE: Have you seen or know of techniques people used to handle-- I think this advice works really, really well when you have a nice guide in your classes. But I worked for four years at a company that does like insurance tech. And one of the problems we were solving is crash detection. And like, Apple, I don't know. They came out with their crash detection model a few years ago. And we kind of laughed because they ran into a problem very quickly that we had seen, was people go on roller coasters and it starts detecting crashes, like crazy.

And one of the key problems in this domain is that you're going to get five car crashes for every million miles of driving. So if you want to scale your data up really big but respect with the distribution shift, like between less than a fraction of a percent of actual data being true positive, we end up training on 8-terabyte data sets that have 2,000 crashes. And a lot of this [INAUDIBLE] does not scale.

PHILLIP ISOLA: So yeah, great comment that if the stuff you care about in your data is extremely rare, then just scaling your data might not be the most efficient way of capturing those rare classes, because you'll be wasting most of your scale on the common case, which is already well understood. And I'll mention this a little later, but data diversity and coverage over the classes of interest, that's the key thing that matters. So scale is kind of a proxy for, actually, coverage over all the different things that matter. And yeah, diversity is the other key word there.

But here, I'm not just saying big data as in a lot of data points, but also-- so if you have this rollercoaster issue, how would you solve that? Well, you would add more information and context into the inputs to say that if I know I'm at an amusement park, because that's in my context, then you should have lower probability of states to crash. And what do language models do now? They're all trying to make the longest possible context. You hear about, like, a-million-length contexts, in terms of the tokens in language models. This is basically trying to achieve the same thing. If you have enough context, then a lot of things become easier and more performant.

OK. So that was all about data, and that was already more than half the lecture. So data is the most important thing in machine learning. So it's good to it's good to focus mostly on that. But models and other things are also important. So let's talk a bit about that. OK. So I think the first principle of modeling is, keep it simple. Why keep it simple? It just pays off multiple times.

Simple models are easier to build, to debug, to share. It's tractable to understand. So we can't do any of this theory and really understanding how these things work and giving guarantees about them unless we keep things really simple. And it may be a little frustrating to be like, oh, why are we proving properties of MLPs? Like, I want to do a more complicated model. But it's because it's easier on the simpler models, and so we should take that as a constraint on what we can actually make progress on.

It's easier to build theories around simple models. But this is the other one. You might think, oh, that's a trade-off. Like, I want a really powerful model. No. Simple models are also the most powerful models. Well, the simplest model that fits the data, that is, by some arguments-- that we talked a little bit about-- going to be the best model. And there's the intuitive argument of Occam's razor. And then there's a lot of formalizations of that with generalization bounds and ideas of optimal inference.

OK. So I would just say-- I've been saying this for years in lectures like this, but I still think it's true. If you focus on simplicity and overweigh that relative to what the average person does and are willing to sacrifice some accuracy, are willing to sacrifice some other things, you'll have an unfair advantage, and your contributions will really stand the test of time.

So what I tell my students is, if method A gets 1% better on benchmarks than method B, but method A is more complicated than method B by some factor, go with the simpler-- publish the simpler method. There's so much competition in research to have these bold numbers, the leaderboard or the benchmark. And if you just remove all the bells and whistles that got you to that bold number, that being the leader, you'll probably have a method which will stand the test of time and be higher impact long term.

So don't fight for that last 1% by adding complexity. Every bit of complexity you add has a cost. And it's usually not worth a little tiny change in accuracy on your problem, unless you're, like, saving lives at a hospital. OK. So anyway, there are some cases where it might be. But keep it as simple as possible.

AUDIENCE: [INAUDIBLE]

PHILLIP ISOLA: Yeah?

AUDIENCE: But a lot of times, how would you know that you're sacrificing--

PHILLIP ISOLA: Yeah. How do you know that you're sacrificing-- like, what is this trade-off ratio between complexity of the method and accuracy on some benchmark? I don't know how to measure that. I just think most people are calibrated too much toward accuracy or performance on the benchmark. And it'd be lovely to have a way of measuring simplicity. But again, we don't really have that. So it's more just an intuition. Yep?

AUDIENCE: Is it good to use out-of-domain data for the training set so they don't have a test set?

PHILLIP ISOLA: Is it good to use out-of-domain data in the training set? Yes. So roughly, again, this principle that trained on as much as possible and put as much information into access possible is, I think, a good principle. And I'll talk about the idea of pre-training in just a second. And we'll see lectures on that a little bit later. But the general trend is, if you have data x y, and z and your problem is about data z, you should train on x, y, and z. Don't only train on z. x and y are strictly beneficial towards z if you're doing things right. Like, there's some kind of data-processing inequality type argument. You could always just ignore x and y if you had the optimal algorithm.

OK. So yeah, popularity matters more than performance. And simplicity matters more than anything. So you want to work with systems that are widely used. That usually means they have to be simple. And it also means that they will interface well with the community and that you won't just be building some amazing palace that nobody ever visits.

OK. So a lot of folks are always going to be looking for, like, but where do I even find the good model to start with? Where do I find the popular standard model? So GitHub, the number of stars or forks on GitHub is a good proxy for quality. So this is a little bit of a sad story, but popularity matters. Don't look at the leaderboard. Look at the number of stars. The stars is more predictive of if it's going to be useful than the accuracy. Of course, this will mean that you're only going to be doing things that are popular. So if you want to do cutting-edge research, where you're really changing, you're innovating beyond the state of the art, this advice does not apply. But if you're just trying to solve a problem at a company, popularity matters more.

OK. So Hugging Face is a great resource for wonderful popular models. GitHub, PyTorch has a lot of repositories. And Papers with Code is the place to go for the leaderboards of the latest, greatest state of the art model. But you probably don't want to do that. You want to look at the latest, greatest one. Look at how many GitHub stars it has. Go down the list until you find one that has 10,000 GitHub stars, and do that one. So the highest-performance model that has 10,000 GitHub stars is like-- that's the rule of thumb.

OK. But you definitely want to start with a pre-trained model for practical purposes. Again, this might differ between theory or you're writing the latest research paper. But if you just want a system to work, this is the advice. So stand on the shoulders of giants. Use pre-trained models. We'll talk about pre-training and transfer learning a little bit later. But you generally don't want to be training your systems from scratch.

And be aware of the flaws and the copyright issues and the biases and all the other problems with pre-trained models out there. But there are some like AlphaFold and AlphaFold 2, which are probably pretty safe to use. There might not be too many negatives, social biases, or ethical issues there.

OK. So here's another little trick that I like, which is to transform your problem into a solved problem-- again, it goes back to the simplicity and popularity are things that matter a lot. So if there exists a solution to your problem, just transform it into that solution. This is a little like in CS theory, where we talk about polynomial-time reducibility. Like, I can show that I can find a polynomial-time method that will convert my problem into a known problem. And this gives me some way of understanding my problem.

OK. So can I find a way of changing a new problem that I don't know how to solve? Like, in this example, I'll give image colorization to a known problem that I already know how to solve, like image classification. OK. So this is a project that I was involved in almost 10 years ago now. You're going to try to predict the color of a black-and-white image, predict every pixel value.

So at first, you're thinking, this doesn't look very much like just a classification problem. I'm trying to do something like regression. I'm trying to predict the color. And I'm trying to predict the color of every pixel, not just one label. OK. Well, we set it up as empirical risk minimization. So find the function f that makes the correct predictions on the training data and expectation. You can predict the colors, and then you can concatenate them with the input image. So you don't have to predict the black and white. It's already in the input. You get a colorized photo.

OK. So how do we turn colorization into-- how do we reduce that into classification? So, first, we say we could just predict the color of the image, right? Yellow, it's a class, right? So just have k different color classes. That would be OK. So how do I turn colors into classes? Well, I can just quantize my space. So I can say, I'll have one class is yellow. One class is orange. One class is blue. Each class, I'll represent with a one-hot vector, just like we normally do for classification problems.

OK. So now, rather than cat, dog, elephant, it will be the blue class, the yellow class. And each box here will be a different discrete class. OK. So now, just like I could predict the label as rockfish, I can predict the label as color. It's the same math. OK. But how do I do that for every pixel? OK. What do you think? How am I to change from doing a whole-image classification to a per-pixel classification? We've already seen it. What architecture should I use?

AUDIENCE: CNs.

PHILLIP ISOLA: CNN, ConvNet. Yeah. OK. So ConvNet just says, I will predict the label of the center pixel of a patch. And if I slide that ConvNet across the whole image, I will be predicting the label, the color, of every pixel in the image. So now I've converted classification into per-pixel color class prediction over a sliding window operator. OK. And so that allows me to formulate my problem as classification, which is generally a good idea. Or we would call it softmax regression for the softmax cross-entropy loss.

OK. So why is this generally a good idea? Well, we have really good machinery for it. We know how it works. But there's another interesting property, which-- OK, I'm not going to have time to go into all the details. But the softmax distribution, the categorical distribution that you predict, is fully expressive over all possible-- so for every pixel, I'm going to predict the color.

And I'm actually modeling a fully expressive distribution over the space of probability mass functions over the discrete vocabulary that I'm modeling. And this is the most general way of quantifying uncertainty or modeling a distribution over a discrete variable. And if I were doing, like, regression, least-squares regression, that would not be the case. It can only represent Gaussian distributions. So this is why classification has this nice property of being more expressive than regression for this type of problem. And discrete classes are also easy to label, easy to talk about, easy for humans to understand.

OK. Another one is that under this representation, we remove the inductive bias about how we have measured distance in the input space because all one-hot variables are equidistant from each other. So the distance is always 1 or 0, is 0 if it's the same class and 1 if it's two different classes. So this removes inductive bias about how we represent target variables.

OK. So here are some good default choices for this year for deep learning a new domain. So turn your data into one-hot vectors. Transform your goal into a cross-entropy classification. And use Adam and transformer to solve it. So just the basic recipe-- this will work pretty well.

OK. But there's a few things that you should be aware of not to use in my opinion. Again, this is a bit subjective. Don't use batch norm. I think that lesson has kind of gotten out now, so I don't know if I still need to say this. But if you're using batch norm, just get rid of it. So batch norm introduces a strong dependency between the batch size, which is one of the cardinal hyperparameters that matters in deep learning, and the learning.

So batch norm is normalizing activations with respect to the statistics of a data batch. So you're introducing this even stronger dependency on batch size. So now your performance will change dramatically when you change batch size. And this means that if I change my batch size, I have to retune my whole model. And I don't really want to do that.

Batch norm has different performance at training time and test time, which is undesirable. Because at train time, I will have a big batch, and I'll take the statistics from it. At test time, I usually have just one data point in processing, and I have to do something else. I can't normalize with respect to the batch. Instead, you use learned statistics.

It makes distributed computing really hard because I can't just shard up my batch onto a lot of nodes in my cluster. I have to communicate information about the different elements of the batch across nodes. So you have this communication overhead. So just use layer norm. That's the standard one right now. Maybe there'll be other standards in the future-- some stuff you can read, if you want, on the problems with batch norm. And I already mentioned that in the Pix2Pix project, we did batch norm on batch size 1 and had a bug. So that's one problem. But that's just for you to read after.

OK. So the easiest way to get your model to perform better is actually not to do anything fancy, but just scale up your data. I already said that. Scale up your model. Make the model bigger, wider, deeper more channels, more heads. And scale your compute. Train it for longer. So just remember that this is like the simple recipe. This is kind of the scale is all you need recipe. And it's not necessarily the entire story, but it is part of the story.

So here's a reminder for me to tell the story about-- when we were working on that colorization project that I mentioned on the previous slides, we were trying to come up with a new model as opposed to just turning it into classification and running a standard convolutional network of the era. And so Richard and I, who were working on this, we went and visited a lab in France for a month that summer.

And when we left, Richard just said, OK, well, we don't have our model ready yet, but I'll let the baseline keep running while we're away. And so we just turned the GPU on. And it started doing this regular thing. It had horrible results at the time. And so we go. And then, like, two weeks later, we check on it, and it's beautiful. And it's better than all the kind of crazy advanced architectures and variants that we tried. So just if you're ever not sure what to do, just train for longer. Go to France. Go to the beach. Relax.

[LAUGHTER]

And a lot of people call this the bitter lesson, or they worry about this, and they think it's diminishing to science and truth. But it's like, truth might be simple and easy. And if it is, all the better. It's, to me, very sweet. So I don't think that that is all there is, but I do think that this is a really, really big part of it.

OK. So there are reasons to work at small scale, because it will allow you to iterate quickly. It will force you to be efficient and come up with clever algorithms that do scale better. But eventually-- I would say scaling is necessary but not sufficient. So it's not that scaling-- I don't really think there's going to be any way of making performant, human-like intelligence without massive scale. That's just going to be a necessary condition in my opinion.

OK. So in my opinion, there's not a hope to solving it with small models and not much data and not much compute. But we'll see. Maybe somebody will come up with that. So a lot of people make their whole system. And then they get it to be really, really performant, and then they stop. But you're only half done when you get there. When you get your system to work and be scaled up, you're only half done.

And this advice comes from the author of *The Little Prince,* which is shown here. So once you get your system working, now you should go back and remove all the nonessential components. Because you probably added a lot of things that, at the time, were important but, once you scaled up and once you got other parts of the system to work, were no longer important. So don't forget to go and remove all the stuff that's nonessential.

So "perfection is finally attained not when there is no longer anything to add but when there's no longer anything to take away." So I think projects should always go-- add, add, add until it works. Remove, remove, remove until it stops working. And cut off there. And then report it.

OK. I'm a little short on time, so I will go quickly over a few slides or skip a few slides to get to some of the final points. Copilots, coding assistants, you should be using these, not on your Psets. So we're going to clarify on the next Pset a little more about how to turn off the Colab's autocomplete. We weren't super clear on that. But it is course policy. You shouldn't use AI assistants in a way you wouldn't use a friend. So you shouldn't have them just write all the code for you.

You can ask conceptual questions and iterate and figure things out with them. But for your final projects and for life in general, you should absolutely use coding assistants. These are just going to get better and better. They have some mistakes right now, but this will be an essential tool in the software engineer's toolkit. And you should be learning how to use them. And this class is about AI, and so we're very happy for you to learn how to use these.

But you should always-- don't replace thinking. Think first, and then ask the LLM. Don't trust the code without understanding what it's actually doing and verifying it and testing So there's good practice for using these. You definitely don't want to just use them blindly without actually having knowledge about how it works. And general advice is, write a very clear docstring, and then autocomplete with a Copilot will work pretty well. And you can prompt it by saying, you're a really smart coder, if you want. But I think some of those tricks will disappear because they're being baked into the new systems.

OK. So, first, when you're going to optimize your models, start with one data point and overfit to that data point, then a few data points. And then finally start scaling up. So scaling comes last. Check your-- oh, actually, this one's kind of fun. So it's good to get used to log numbers, because most of the time, in deep learning, we're talking about log probabilities. Or like, the cross-entropy loss is the average log probability of the data under the model.

And so get used to log probabilities. Know, is a negative number OK? Is smaller good, or is bigger good? What if it comes out 0? What does that mean? So here's a few log probability numbers just to memorize. So what does negative 0.69-- if you get that number as your loss, is that good? Is that bad? That's just log of 0.5. So if you have a binary classification problem, you'll often see you get negative 0.69. You're like, OK, it looks pretty good. No, that's chance. That means you're outputting equal probability to both classes for a binary classification problem with equally weighted classes.

OK. What is log of 10%? Like, 10%-- we often have 10-way classification problems as our standard homeworks. Negative 2.3 means you're at chance. You're outputting 10% probability over all data points. OK. So with regression loss, like a least-squares loss, it better not be negative, because the minimizer of least squares is 0. The minimizers of log likelihood is negative infinity. So it's good to get used to these and know when to look for bugs. If you see negative 1,000 on a regression loss, you know there's a bug.

OK. Always tune your learning rate and, often, your batch size. OK. I just want to-- let's see. Oh, I have a lot. Well, you're going to have to read some of these slides after. Yeah, sorry. We got a little late start.

OK. Another one that is kind of fun is, use exponential moving averages to smooth things out. So what is an EMA? An EMA is Exponentially Moving Average. It just says, take whatever value you have. And over time, you will mix in the new value with the previous values multiplied by some decay factor. So you're exponentially decaying the contribution of your previous values with the new values.

And people use this all the time to trade off between averaging across space-- so let's say I have a batch of gradients. And I'm going to take an average over all my-- like, the sum of my gradients is going to be my estimate of the total gradient. And you can replace averaging across space, across batch dimension, with averaging across time, across iterates of your learning using EMA. So it's a trade-off between time and space.

And whatever quantity you have in deep learning-- gradients, activations, data, weights, even target predictions-- you might want to try putting an EMA on that. You can, in PyTorch, just have some wrapper that says this variable will not be its value but be its value summed up over all the previous values. And then it becomes kind of the EMA version of that variable.

OK. So, yes, I packed way too much into this lecture. OK. This one, I want to point out too. OK. So there's all these pieces of advice and little tips here. But if you're going to make some nice dish for dinner tonight, which spice should you use? So somebody might say coming from Google or someone might say, oh, cayenne pepper is all you need. It's just the best. Just use that. OK. Is that right? Is cayenne pepper all you need?

No, it's not quite like that. So attention is not all you need. Attention is one thing that works pretty well and has certain effects. ReLUs are nice. But it's really like cooking to me. It's very multimodal in this such landscape as a function of all these things. You want to find the right combination of spices that solves your problem. Every regularizer you add is going to have a certain effect. And they'll overlap. So you have to have some regularization.

Do you need to have skip connections? Well, yes, you do if you have a really deep architecture. But maybe you don't if you have an optimizer which somehow doesn't have vanishing gradients. So you have to get the right combination. And there's many combinations that are all equally valid. So you can have a nice, spicy chicken dish with cayenne and peppers on it, or you could have a nice sushi with miso and soy sauce and things like this. And they're both good.

And you just want to learn which ingredient to use to have which effect. Some ingredients regularize. Some ingredients increase capacity. Some ingredients make optimization faster. So you just have to get the right combination. And don't overspice. Don't overrely on one. OK. So I think I'll end there. The rest of the slides have various tips that you can read on your own time. I hope that they're self-explanatory. And I will see you next week.

[APPLAUSE]