[SQUEAKING]

[RUSTLING]

[CLICKING]

SARA BEERY: So why are we all here? Deep learning has clearly been exploding in society. Machine learning generally is something that, when I started studying it about 13 years ago, didn't work, and now it works. So how many of you here in this room used AI in the last week?

Yeah, almost everybody, probably everybody. And we're seeing massive advances in everything from AI-assisted text generation to 3D reconstruction, things like NeRF, things like AlphaGo, generating images, helping with writing code, playing games. This has really started to touch almost every dimension of society, and it's definitely very exciting to see. And I think that hopefully, all of us will learn a lot more about what that means in this course.

So what is deep learning? First, I would argue that one necessary component of deep learning is the idea of neural networks. And these are a class of machine learning architectures that basically use stacks of linear transformations interleaved with pointwise non-linearities. And these have really been a building block for a lot of the progress that we saw in the last slide.

But the other really important dimension of deep learning is the idea of differential programming. And this is essentially a programming paradigm where we parameterize parts of the program and then let gradient-based optimization tune the parameters to basically optimize that program or find some of, at least, local optimal for that program. And these two things together is what we posit make up deep learning. And we're really going to dig into both of them in this course.

So our philosophy for this is that breakthroughs in deep learning have been driven by a mixture of theory and practice, and both dimensions are really vital for future progress in the field. And so along those lines, this course should hopefully provide both a theoretical grounding in important deep learning building blocks, as well as give you practice implementing, understanding, and using those blocks.

OK, there's still a lot of people in the back who are packed in. I don't know if there's any way to come further forward. If not, we'll do our best.

So the class's coursework is going to be 65% problem sets. You'll have five psets. Each will be about one to two weeks long. And then there will be some combination of pen and paper or, more realistically, maybe Overleaf, as well as code that you will need to do and submit. And then 35% of the grade will be a final project. And this is a research project that's focused on deeper understanding of some of the topics that we cover in the course.

You'll be asked to propose something for that project. So there'll be an initial project proposal. And then the final project will be a blog post that's going to demonstrate novel experimentation and visualization.

And the hope here is that I don't know, many of you-- how many of you are machine learning researchers who have already written a machine learning paper, as first author, or anywhere on the author list? OK, so some. These days, when you write a machine learning paper, more often than not, you also have to write a blog post about that paper. And that's intended to be catchy, make people understand the material, but really get them engaged.

And so one of the reasons that we have this as the format of our final project is to reflect on the fact that this is the reality we live in today, that in machine learning research, being able to write a blog-style description of the work that you're doing and not reduce any of the technical complexity or the information that's given across, but just do this in this slightly more visual and maybe even interactive format is a really valuable skill, and one that we hope this can help us think about. And we'll allow groups, but only up to two.

And another important component-- we are not able to provide compute. We may be able to provide some very limited compute, but that's TBD. So do not plan your final project to be something where you need to get state of the art by crunching a huge architecture on a bunch of huge data sets, because it's just not going to work. You're not going to be able to outcompete OpenAI on this research project. [LAUGHS]

So this does not mean you can't do good research. And it's important to get creative here. And I think it's actually a really valuable skill, particularly as certain types of large-scale deep learning become more and more centralized, to think about how you can still do impactful machine learning research in a way that is not just "bigger is better."

All right, any questions about coursework or logistics? Yeah.

AUDIENCE: Is the final project expected to be a mix of practical and theoretical research?

SARA BEERY: So the question was, is the final project expected to be a mix of practical and theoretical research? I think it depends. It depends on what you want to propose as your final project. But I think we're pretty open. What we want is for them to be innovative, but that innovation can be in an applied sense, or it can be in a more theoretical sense.

OK, so I'm going to do a really high-level overview of the schedule, just to give you a rough sense. So today, we're really covering basic interest stuff-- course overview, a rough introduction to what neural networks are and basic building blocks, and, particularly in the context of signposting, what we expect you've seen before, because this is not an intro to deep learning class. This is advanced graduate-level deep learning.

And then we're going to go into how you actually train a neural net, then approximation theory, some different architectures, things like grids, graphs. Then we'll hear from Jeremy about scaling rules for optimization. And then we'll look at generalization theory, more architectures, getting into transformers. We'll do a really fun lecture by Phil called the hacker's guide to deep learning that I think is quite useful.

And then we'll talk about memory. And then we'll get into representation learning. So there, we're going to look at reconstruction based and then similarity based and then theory of representation learning. Then we'll have a little bit of a series on generative models, the basics, how representation learning interacts with generative modeling, thinking about conditional models.

Then we'll talk about generalization, and particularly out-of-distribution generalization. We'll touch on transfer learning, both from a models and a data perspective. We'll hear about large language models. We'll have some guest lectures towards the end of the semester, and those are TBD. Then we'll hear about scaling laws, automatic gradient descent, and potentially the past and future of deep learning, though I think there's some discussion as to whether that's going to be a different lecture.

And then, I think kind of importantly, right towards the end, we will have class-wide project office hours where we'll have a bunch of the TAs and the instructors all come, and we can do open discussion around projects before they're due.

So the next thing is that we're going to host PyTorch tutorials. This is intended for if you're not familiar with PyTorch, and you think you need a refresher. And we strongly recommend that if this is the case, or if you even think it might be the case, that you attend one of the two PyTorch tutorials we're going to offer next week.

So the question is if PyTorch is a requirement, or you're allowed to use any other frameworks. So you're probably welcome to use any other frameworks in your final project, but some of the problem sets specifically will have code that's already built into PyTorch, and you're filling parts of that in. So unless you want to rewrite all of it in JAX or something, which more power to you, I would recommend that you definitely make sure that you are familiar with PyTorch. OK.

Cool, so I'm going to talk about course policies. And this stuff is important. And it's also all on the web page. But I think essentially, what we want to touch on is what our collaboration policy is first.

So everyone is required to write an individual pset. So when you're actually working on your problem sets, do not work on something together and then both submit the exact same thing. You can work on the actual-- the content, you can discuss that with peers, TAs, and instructors, but the problem sets you submit need to be your own independent and individual work.

And that also applies to the code that you write. So you can discuss the code with others, but you need to each write your own code separately. There was some confusion about this in the past. Just make sure that you're not taking the code someone else wrote, running it, and then submitting a pset on it.

You should absolutely not copy or share complete solutions, and you also should not ask someone else, hey, this is my solution, is it right? And you shouldn't do that in person, and you also shouldn't ask, Is this correct? on Piazza or Canvas. So again, the idea is, it's your independent work, your independent thought. And I think it matters more that you've gone through that process independently than actually getting it correct because you'll learn from the feedback on your psets better if you submit something that you came up with yourself and get feedback on it than if you copy somebody.

And then if you work with anyone on the pset other than TAs and instructors, we ask that you write their names at the top of the pset. So if you have a study group, you're all working together-- we don't care if this is 10 people. We don't care if it's, in this class, 100 people. But we just want to know who was working together. I'd be impressed if you managed to come up with a 100-person collaborative group.

So AI assistance-- we take the exact same policy about things like ChatGPT or other AI assistants that we do with humans. So it's a deep learning class. You should be trying out the latest technology. The idea is not that we're teaching you to ignore what's happening.

But we do want you to treat them like a human collaborator. Don't ask them to answer the question for you. Just try to imagine from an ethics perspective that this is like a peer in the class. So don't say, what is the solution to this? Don't say, write the code for me, because you wouldn't go to your friend and say, please write my code for me.

But you're super welcome to use them as a discussion partner. Ask them questions that are contextual. Go back and forth. That's all fine.

But don't treat AI collaboration as if they're not a human collaborator. Think of it that way first. Is this something I would ask my friend in the class?

And then just like you can come to office hours and ask a human questions about the lecture material, clarifications about questions, tips for getting started, you're welcome to do the same with AI assistants. Again, you're not allowed to ask an expert friend to do your homework for you. Don't ask AI to do your homework for you.

If you're ever unclear, imagine the AI is a human and apply that same norm. And if you work with any AI on a pset, similarly to the fact that we're writing with humans we collaborated with, please write down which AI and how you used it. And, of course, this is a bit vague, but again, just kind of honor code.

Any questions about AI collaboration, or is that reasonably clear? Yeah.

AUDIENCE: I had a question on the last page. So can we show our work to the TAs to [INAUDIBLE]?

SARA BEERY: Yes. You're very welcome to show your work to the TAs. The TAs are not going to tell you how to answer the question, though, but they will work with you to help you hopefully figure it out on your own.

Why are we here? [LAUGHS] What's the goal? So what do we want to do? Why are we all interested in things like deep learning?

One perspective is that we're interested in modeling complex phenomena in the real world. What are complex phenomena? Well, you interact with them every day, things like natural language, images, or visual data in general, videos. DNA, ecosystems are complex phenomena. Climate change is a complex phenomenon.

Why is it hard to model complex phenomena in the real world? They're complex. [LAUGHS] And so maybe one existence proof for deep learning as a solution to modeling complex phenomena-- possibly, we could talk about the human brain. All of you are here at MIT, which means you're pretty good at modeling complex phenomena in your brain, I'm guessing.

So first, today, I'm going to just briefly go over how we got where we are today. This is a brief history of the field of AI, deep learning in general. And then I'm going to go over what we expect you have already seen before.

It's OK if you haven't seen these things before, but we would expect you then to go and brush up on them. So if there's anything that I talk about, and you're like, oh, I have never seen that, we expect that. That's something you can go look at some of the OpenCourseWare courses at MIT, go find lots of resources online, and pick it up before you go into this class. And then I'll also talk about what we actually cover in the class.

So a brief history of neural networks-- and I think a fun way to go through this is looking at this on a plot of enthusiasm for neural networks over time. So if we go way back to 1958, Rosenblatt introduced the perceptron in *Psychological Review.* And the perceptron was the first neural network, arguably.

And what this was intending to do was recognize or categorize images. And the idea was essentially that you could take some combination of representations-- in this case, particularly pixels-- sum them, put them through some non-linearity, and get out a categorization. And really, we'll see throughout the course and even today, this idea still forms the building block for most of the deep learning that we do. And these perceptrons are components in many, many, many different types of capacities within things like ChatGPT.

When this paper came out back in 1958, people were incredibly enthusiastic. It made waves. And then in 1972, Minsky and Papert wrote a book called *Perceptrons, Expanded Edition.* And this book takes a very critical lens to the idea that a perceptron could be seriously considered a model of the brain. And it also carefully and mathematically describes and characterizes the limitations of what a perceptron can model.

And it did so, so carefully and so critically that all of a sudden, enthusiasm for machine learning really took a dip. So in the '70s, we were not excited about neural networks.

But then in 1986, this book came out called *Parallel Distributed Processing.* And this was fundamental because it introduces the concept of backpropagation. And this was really the thing that enabled multilayer perceptrons, so not just a single perceptron, but actually multiple perceptrons stacked on top of each other to actually be reasonably trained, reasonably fit, this concept of being able to backpropagate a gradient through multiple layers of perceptrons.

And what this enabled you to solve were things like the XOR problem. For the first time, you could reasonably solve categorization problems that were not separable from a linear classifier. They pointed to backpropagation as a breakthrough. It led us actually handle things where you need multiple different boundaries and these nonlinear boundaries to be drawn, which is only possible if you have multiple layers of a perceptron. You'll never be able to solve the XOR problem with a single-layer neural network.

And everyone got really excited again. So in the '80s, yet again, machine learning was something that people were pumped about, and particularly this deep learning neural network perspective on machine learning.

In 1998, Yann LeCun put out convolutional neural networks. There's an awesome demo that he has on his web page. I'm sure many of you have heard of Yann LeCun. And this seems like it would mean that everyone was really excited about neural networks again.

But if you go to NeurIPS in 2000, this is the premier conference on machine learning. It evolved from an interdisciplinary conference, which was on basically the science of the brain and machine learning to specifically just a machine learning conference. If you look at that conference in 2000, the most predictive words in papers in the title for acceptance were "belief propagation" and "Gaussian." And the title words most predictive of paper rejection were "neural" and "network."

[LAUGHTER]

So this is what we called the AI winter. And one of the reasons it was called the AI winter is because even though we had the theory, we had the building blocks, we didn't actually have the ability to train them efficiently. We didn't have the right programming perspective, and we didn't have the right hardware.

But then in 2012, Alex Krizhevsky and his co-authors published AlexNet. And AlexNet was also another convolutional neural network. But importantly, Alex Krizhevsky was a brilliant programmer, and Alex Krizhevsky figured out how to program GPUs, graphics processing units, that have been developed to handle massive-scale parallel multiplications for graphics, for video games, and for high-resolution images. But he figured out how to program those and repurposed that hardware for the training of neural networks.

And I can't tell you how much of a shot heard around the world this was in our field. But this, for the first time, outperformed every single other possible method that was out there on ImageNet. ImageNet-- how many of you have heard of ImageNet? Yeah. ImageNet was the first very large-scale data set of its kind.

And I actually think that this is an important component, what made machine learning start to work. And I would argue that, of course, it was the theory and the programming, the ability to actually efficiently fit these things.

But the other really vital component was the data. Machine learning does not work without large-scale, curated, labeled data in many capacities still, though we're getting better and better at self-supervised learning. But so all of a sudden, we were back. Everyone was enthusiastic again.

And actually, if you look, these were hype cycles of about 28 years, both of them. And so the question is, where are we going to be in 2028? So it's 2024 now. It's 2024, so in four years, where are we going to be?

So one argument would be, OK, we're in another hype cycle. But I would actually argue that that's probably not true. Another one would be that we're just way off the deep end in terms of enthusiasm, and by 2028, we'll be in some-- out in the stratosphere.

Or possibly we're doing something like this. Maybe in 2028, we're going to hit some new plane of enthusiasm, and then we'll start to realize limitations, and we'll oscillate again. Yeah?

AUDIENCE: Could AI take over society by 2028?

SARA BEERY: [LAUGHS] So the question is, will AI take over society by 2028? My answer is, let's see. [LAUGHS]

So what's deep learning today? What are the components that make it up? First, I think a really important one is Autograd. These are coding languages like PyTorch and TensorFlow that give us a really clever way to do things like implement the chain rule in software, making good use of available hardware.

And this has enabled us to do things like approximate the gradient of any function, even if it's an approximation. These are a really important component of deep learning. And if you do anything with deep learning, I doubt that you're doing what Krizhevsky did. I doubt that you're programming these things from scratch. You're almost certainly working with existing programming languages that are built for machine learning.

Its billion-plus data point datasets, things like LAION. So that scale of data that I talked about before, this is an incredibly important component of all of this. Parallel training on thousands of GPUs-- again, this is what we're looking at today-- billion-plus parameter architectures, and this number just keeps increasing, million-plus dollar training costs. So, of course, it's not public knowledge how much it costs to train something like GPT 4.0. But I would bet it's definitely in the many millions.

Shockingly good results-- I don't know about any of you. I mean, I've been working in this field now for a while, and I think we've been seeing results recently that I didn't anticipate seeing in my lifetime from when I started, back when things didn't work. So that's been both exciting to see, and also sometimes, I think, for many of us a bit overwhelming, because you're like, oh, gosh, the problems I thought were problems aren't problems anymore. So what are the problems?

Of course, I work in the world of-- I work a lot on biodiversity loss at a global scale using machine learning. And I would say that biodiversity loss is definitely still a problem, and one that machine learning is sometimes helping, but often also hurting, because the carbon costs of actually training these models is significant.

But all of these massive things-- massive isn't necessary. So things like Stable Diffusion are actually pretty lightweight and were incredibly impactful. So things don't always need to be big to be good.

And then I think another really important component of deep learning today is the idea that it is primarily an open source community where we have what we call modular reuse. So people take weights that were trained by one person with one architecture. They might take that and use that entire thing as a module within another, larger system that has many different components that are being pulled from different things.

Now, I think that this open source perspective is really something that's driven a lot of progress in machine learning. But also one of the things that we have been seeing in the last few years is that things are less and less likely to be open source. You might have an API, but you aren't necessarily going to get the weights and the architecture for something like GPT 4.0.

So for the rest of the lecture, this is the signposting I'm going to use. Looking back, this is what we expect you've seen before. The green looking forward, that's what we're going to cover in this class. All right, pretty straightforward.

I expect you have seen gradient descent before you come to this class. So gradient descent essentially is the idea that we're trying to optimize some cost function. And the way we're going to do that is we're finding some minimum over a set of all of our training data, some minimum set of parameters, where then the loss is actually minimized for all of those different data points.

So what this actually looks like in practice, you're trying to find some theta star, which is like your optimal set of model parameters that minimize your cost function. And so maybe you start somewhere random, and then you calculate your gradient. Now you update the weights of that model in the direction of the gradient, and you step through, et cetera, et cetera.

And I think many of you have probably seen this before, and we'll talk in an upcoming lecture, cover some of the different dimensions of things like stochastic gradient descent, et cetera. But we're assuming that you've seen this idea before, probably in intro to ML lecture, for example.

Covering in this class, we'll go into a little bit more detail, backprop and specifically this idea of differential programming. So what does it actually look like to build programming languages that are structured around the idea of optimality and optimizing through gradient descent for different components of that architecture? And I think one of the things that's fun is thinking about what are you actually optimizing for, and what are you pushing the gradients through to?

So here, one iteration of gradient descent is essentially something like this, and we have some learning rate. And all of that, we're going to go into in a lot more detail next Tuesday where we're going to be talking about, where does this actually come through, how do we think about what some of these choices are, and how we actually build out the learning structure for backprop.

I do expect that you've seen things like multilayer perceptrons and nonlinearities, things like ReLUs, before. So computational neural network is often some form of vector in, vector out. And this arrow that goes from vector in to vector out is a lot of what we're going to be talking about in this class. What computation does that cover? How does that actually build out?

So one cool thing about neural networks is that they often are reusing the same simple computational units over and over again. So there's often a lot of the same kinds of computation that are being repeated and reused.

And what these actually look like, these building blocks that are being used over and over again, they're often some combination of a linear layer. So here-- and we will be sticking with this type of notation for the rest of the class-- what we call z sub j, so some component of this output representation in that linear layer, that would be a sum over i of some weight times each of those input components-- basically, pretty simple.

And that can also be written essentially this way, where you have some set of weights, and then you also incorporate a bias term. And this is a term that does not actually take in any of those input components. It's just a bias.

And that can also be written in a vectorized notation, which makes everything much more simple. So again, just with a focus on the notation, that output unit z sub j is x, that input vector, transposed times w sub j, so the set of weights that correspond to that output component and that bias term. And then what we consider theta-- so that's all the parameters of the model-- that's the set of all the weight terms and all the bias terms.

So what actually makes a neural net a neural net is the fact that it's not just linear combinations of inputs. You also have some sort of nonlinear function that is mapping your outputs. And that's something like g sub z. And in this case, that's something like a pointwise non-linearity. So that's an important dimension that these non-linearities are usually pointwise.

So one possible non-linearity would be this one. g sub z would be 1 if z is greater than 0, or 0 otherwise. Is this a good choice of non-linearity for a neural net?

AUDIENCE: No.

SARA BEERY: Why not?

AUDIENCE: It's not differentiable?

SARA BEERY: Yeah, so it's not differentiable, exactly. And why is the fact that it's not differentiable not good? Yeah.

AUDIENCE: Because it hinders backpropagation.

SARA BEERY: It hinders backpropagation. Why?

AUDIENCE: It makes the [INAUDIBLE] set harder [INAUDIBLE].

SARA BEERY: Yeah, so the directionality. If you're anywhere on this graph, you don't know which way to go, essentially. The gradient is zero. So you can try to take a gradient ascent step. You're not going to move because the gradient is zero, exactly.

So this roughly is called a perceptron, this linear input plus a pointwise non-linearity. And you actually can arguably do linear classification with a perceptron. So if you take a really simple version of this where your input is just two-dimensional, x1, x2, and you have some w1/w2 learned weight terms that you use to combine to get z, which is your hidden unit, and then you map that via a function to y, you then define z as x transpose w plus b. And y is now some function g of z.

What this looks like for any set of values will be something like this, where you essentially have almost a ramp. It's a clear and standard gradient across.

Why would it look like this? So why is it smooth and reasonably straight?

I maybe didn't pose that question super clearly. So essentially, the reason that mapping it looks like this is because it's a plane. So any two-dimensional version of this is going to be a plane with some orientation. And then very straightforward, actually, to turn that into a linear classifier, because all you're doing is taking some threshold on that plane.

So even a single-layer neural network can perform linear classification because now this threshold, that's basically what that stepwise non-linearity is giving you. It's giving you something that is thresholded at some value. And so even with something quite simple, you can do classification.

Now, that's assuming that your data is linearly separable. So if you have training data like this, which is linearly separable, and now you want to find some optimal set of weights and biases that will minimize some loss term where now, again, this loss is the difference between the true value-- so here, the true value is 1 or 0-- and the functional mapping of the inputs to something like that value.

If you have something like this, this is what we'd call a bad fit. You have 7 misclassifications. You're wrong about more things than you're right. And all of those red values, those are going to give you a high value in your loss function.

Or you could find something that's like an OK fit, less misclassifications. And maybe this is something that your model updates to after a single gradient step. Your decision surface is linear, and now you're fiddling around with what that classifier is-- basically, which values of weights and biases you need. And then maybe after a few steps, you would find something that is a good fit, where you're not actually misclassifying anything.

So it is possible to do classification of linear things, linearly separable things, even with something like a perceptron that's doing these really simple thresholding-based non-linearities. But like we said, this actually would be really difficult to learn.

So instead, maybe you would try something like a tanh. So this is another pointwise non-linearity. And it was one of the earliest ones that people looked at. The things that are nice about this is it's bounded between minus 1 and 1, so none of the values are super huge.

And you get saturation of the gradient for super large or super small inputs. You're not getting explosions necessarily. But also, if you start out somewhere that's really far from the center, you don't have a lot of training signal because we're getting close to 0 in our gradient. And so those gradients do go to 0 as we go to infinity in either direction. The outputs are centered at 0.

So the tanh of z can also be written as 2 times the sigmoid of 2z minus 1. So let's look at what that sigmoid function looks like. So in this case, this is a sigmoid, 1 over 1 plus e to the minus h or e to the minus-- the notation is wrong. It should be minus z.

This can be interpreted as the firing rate of a neuron. That was one interpretation of it initially. It's bounded between 0 and 1, so it's not ever negative. Again, you have saturation of that gradient for super large or super small inputs. And all those gradients go to 0.

And the outputs themselves are centered around 0.5. So this is kind of poor conditioning. It's slightly biased. In practice, we don't actually use this.

So instead, in practice, what we often use is something called a rectified linear unit, or a ReLU. And this is just the max of 0 and z. And this is unbounded on the positive side, so it can go all the way to infinity, which can result in, for example, sometimes things like exploding gradients, because the values can get very large. But it's super efficient to implement. Essentially, the derivatives are just that step function.

And it also seems to help speed up convergence. So in that Krizhevsky paper, they showed that they got something like a 6x speed-up using a ReLU over using something like a tanh in terms of converging that model and learning.

The drawback is that if you're strongly in the negative region, the unit's what we call dead. There is no gradient, and so you essentially have this gradient collapse, where you can no longer learn from that component of the vector. But this is our default choice, and it's widely used in current models. There's lots of slight tweaks to this, but this is really, really common.

So then the last thing I want to talk about in this section is stacking layers. So we talked about a single version of this, where you have your linear projection, and then you have a non-linearity. But now you can take that, and you can just stack another one on top of it. So now both z and h are hidden units. They're somewhere in the middle of the model. We don't necessarily-- they're not seen at the inputs or the outputs.

And essentially, then, you can also remember that all of these things are stacked on top of each other, almost in parallel as well, because all of these inputs are mapping to the different components of the output vector. And so you have weights for all of those different dimensions.

So this means that h, our output of the first layer, is going to be g, that pointwise non-linearity, applied to now instead of vectorized w transpose x sub j to give us the individual j-th component, you can actually just think of this now as a matrix multiplied by a vector, all of the weights for all of those different dimensions multiplied by the vector x plus the vector of all your biases, b1. So now you have this cleaner notation.

And then now y is again g, that same pointwise non-linearity applied on top of your outputs of your first thing, but now with h, multiplied by that vector of all your second-layer weights and added with your second-layer bias term. And so now, your set theta, your model representation, is the set of all of the different weights from all the different layers, as well as all the different biases in all the different layers. Oh, thank you. Oh, yeah. Thank you.

AUDIENCE: Based off your application, how do you choose your activation function? What are the parameters of what might be considered a good model or, I guess, a good activation function based off of what you want to use your model for?

SARA BEERY: So you're asking, in practice, are there any best practices for choosing the non-linearity based on different types of data. I think in the literature, there's maybe-- you can see it coming out. But often, people really do use ReLU quite a lot. But I wouldn't say that there's any great general rules of thumb there.

And often-- yeah. I would say there's different things that people have converged upon based on experimental success. So you'll see in the literature like, oh, yeah, everything in this line of work on these data sets with these things, they're always using one or the other component. And then people build on that.

Sometimes that's called grad student gradient descent [LAUGHS] because you're essentially writing research papers and learning from those research papers what parameters in terms of things like your activation functions make sense. But yeah, I don't know that I have great rules of thumb of, oh, for this type of application, you should always use this or that. Do you?

AUDIENCE: Yeah, it's not a hard science.

SARA BEERY: Yeah.

AUDIENCE: But it's usually about what makes the network more trainable rather than trying to model a certain kind of data. There are a couple of examples, like using a sinusoidal non-linearity to pick out some for you, if you'd like. There are a few examples that do but usually not. And [INAUDIBLE] does deep learning has a great answer to that question.

SARA BEERY: Yeah. So what Jeremy was saying is basically, it's not a hard science. We don't have theoretical guarantees of this is what you should do. But in practice, there are-- yeah, there's counterexamples to that.

So if you know, for example, that your features are sinusoidal, it maybe makes sense to try to pick out Fourier representations. You're working in the Fourier space, then maybe you should use a sinusoidal activation. Anyway, but yes, I wish that we better understood it. And that's, I think, why we need a lot more research on the theoretical side of understanding, how do we actually choose these things appropriately instead of maybe learning everything from scratch or doing it experimentally, burning all those trees?

Cool. So the parameters of our model are now all the different layers of weights and all the different layers of biases. And so like we said, you can do linear classification with a perceptron. But the fun thing is-- oops-- you can do non-linear classification with a deep net, even if that deep net is just a two-layer perceptron. So here, now we've built a double-layer model, still only in two dimensions. So now we have mapped through essentially just this simple non-linearity in between these two different layers.

Now if we look at what this looks like, that first layer is giving us one kind of ramp. The second layer gives us a different ramp because it's a different linear model. And now if you do this non-linear combination of them, essentially what you get out is you can actually map-- it's almost like think about the intersection of one ramp with another ramp. You get a kind of triangular, almost a pyramidal component that will be higher than everything else. And so now you can still take a simple threshold of that, and now you can get out a non-linear classification.

And, of course, you can expand this into many more dimensions. But this is, I think, some nice, simple intuition building as to why you start getting non-linear models when you're taking linear combinations, but with these non-linearities.

Cool. So another-- so we expect that you've seen a lot of that before. But moving forward, one of the things we are going to cover in this class is what we can approximate. So what do we know about what is possible to approximate with deep learning?

So one dimension of that is representational power. So you have a single layer. You can do any linear decision surface. If you have 2-plus layers, in theory, you can actually represent any function.

And this is assuming, of course, that you have some non-trivial non-linearity between those layers because if you don't have that non-linearity, it turns out that a linear layer and another linear layer, this is just a linear combination. It's still linear. So you need that non-linearity.

But doing that, you can actually approximate anything. And one way to think about it is, given some amount of capacity, some number of different dimensions that you can do, you can actually approximate any function. This is a rough, hand-wavy intuition thing. But as you're shrinking the size of these different vertical things, you're getting something like a Riemann sum. You can approximate any integral as long as you have enough capacity in terms of the number of small things that you're building up.

So this gets at basically one of the issues, which is efficiency. So in theory, you could approximate any complicated function with a sufficiently wide two-layer network. So this means not two dimensions, maybe thousands or millions of dimensions.

But in practice, that's actually very inefficient. So a narrow deep model, maybe something that only has much fewer dimensions in terms of the size of each vector or each input and output, but now many, many, many more stacked layers, you might be able to, with a lot less parameters, approximate the same complex function. In practice, we do find that's true. More layers helps. And we'll get into this a little bit in lecture 3, which is about approximation theory, the theory of what it is possible to approximate.

And then the other thing that we're going to cover in this class is architectures. So when we talk about deep networks, really, often, we're talking about something that's roughly like this, some cascade of repeated simple computations, each of these linear and then linear models with a non-linearity. And then with every increasing layer, we're getting more abstracted representations. You're getting a higher capacity model.

And this means it can form more complex and abstract calculations. Whereas a shallow network, if it's not sufficiently wide, it could maybe only efficiently compute simple things like maybe finding edges, a deep network could efficiently compute harder things like recognizing this clownfish or translating English to Chinese.

And so this classification of complex things, essentially, your whole function f of x is now this stacked or almost chain-like mapping from your final layer, which is then a functional mapping of your previous layer, and in and in and in and in, all the way back to that input. And there's a lot of different ways that we've learned to build these mappings for different types of problems and based on the structure of the data we're trying to model. And we'll get into that. We'll talk about CNNs, GNNs, transformers, and RNNs, so convolutional neural networks, graph neural networks, transformers, and recurrent neural networks.

And then the other thing that we're going to be talking about in this class is when and why we can generalize. So why do deep nets generalize? Essentially, deep nets have so many parameters, they could just act like lookup tables. They would just regurgitate the training data.

But instead, they seem to learn rules that generalize. And this actually defies classical theory. So the classical theory basically says that if your model is overparameterized, it will overfit. There's this capacity versus risk or error curve where if you don't have enough capacity, you're going to underfit the model. And then with too much capacity, you're going to overfit.

But in practice-- and this is from this paper "Double Descent"-- what we see is that in this modern interpolating regime, we go past that point of being overparameterized, and we're actually then able to actually pass some interpolation threshold or ability to have enough capacity in the model to interpolate between the data points. We're actually able to then potentially get even better, even though the models are massively overparameterized. Yeah.

AUDIENCE: What do you mean by capacity?

SARA BEERY: Capacity is the number of parameters in the model. So that's both the width and the depth, basically how many values are captured in those model weights? Yeah. Yeah. Yes, thank you.

AUDIENCE: And how does that relate to the size of the data samples that you're passing?

SARA BEERY: Yeah. So that is not a dimension on this plot, but it definitely is also related. So if you have-- but it's related in a different dimension. If you're learning from fewer data points, it's very easy to learn to maybe overfit to those data points. So you could think of that honestly, in another way, maybe as being a different dimension of capacity here, where it's like almost the capacity of your training data. But, of course, there's not a direct translation.

But so if you don't have enough training data, you also can't learn to interpolate between those data points because you don't have enough coverage of the data distribution. I live in that world a lot. We often don't have enough data to learn a general representation without a lot of handy tricks.

We will talk about some of those different tricks in this class, some of the different ways that we try to convince models not to overfit to data and try to learn to generalize beyond maybe what is captured in the data. But I think that dimension is really important, and it's one of the reasons that none of this was possible until we started building big enough data sets, where you had enough coverage of the space that you wanted to model in your data to be able to start interpolating between them. Yeah.

AUDIENCE: Can you just clarify the difference between overfitting and overparameterized?

SARA BEERY: Oh, yeah. So, I mean, I think this paper would be great to read to be able to really get into it. But essentially, the idea here is that in the previous literature, we said that at some point, if you had too many parameters, you were going to overfit and learn a model that actually generalized worse because based on the amount of data that you had, it was going to-- there's a classical example where you have three data points. And if you try to learn a 12-dimensional function, it will learn something like this to fit those data points perfectly. But actually, it's not likely to generalize because it's basically too spiky or too peaky.

So that's the idea of overfitting. But this idea of overparameterized is basically instead of saying underfitting and overfitting, now we're saying that the new paradigm is underparametrized and overparameterized. And what they're basically saying is that you maybe can't be too overparameterized, though this perspective is one that doesn't take into account resources. So there's many, many people and places and places that are at MIT, but even actually in this class, where we don't have infinite compute resources. So actually, what you really want is some optimal point on this curve where you get good performance, but you're not maybe using resources that you don't have, super, super large memory GPUs. Yeah?

AUDIENCE: When you gave the example of using two very wide layers versus multiple narrow ones, [INAUDIBLE] there's a rule of thumb about which is better for about the interpretability between the two?

SARA BEERY: Yeah. OK, we're definitely going to get into this a lot more in some of the later lectures, these choices between breadth versus depth in models. But at a very high level, your intuition, I think, that you're getting at is correct. If you have only two layers-- actually, maybe not, because if you have two almost infinite parameter layers, really, really high dimensional space is hard to interpret. If you have lower dimensional space over many, many different now stacked versions of these layers, it's also hard to interpret.

So I would say like in ecology, for example, people often still use just generative additive models of a bunch of different linear things because they need the interpretability, or they feel like they do. And anytime you get to larger numbers of parameters in either dimension, that interpretability gets more difficult.

AUDIENCE: I was just going to add that even in, let's say OpenAI, it's a closely guarded secret what recipe of width versus depth that they use in GPT whatever. If you can come up with a really scientific way to answer that question, you could go and get a job there and--

SARA BEERY: Make 7 figures. [LAUGHS]

AUDIENCE: [INAUDIBLE]

SARA BEERY: Yeah, so what Jeremy is saying is, there's no perfect prescription or recipe for width versus depth and what's optimal. Yeah.

So the simplicity hypothesis in classical theory basically says that big models learn complicated functions and overfit the data. And the emerging theory is that deep networks actually learn simple functions that generalize. And we're going to talk about this a lot more in a theoretical and a more experimental lecture around generalization, both in and out of distribution.

So again, what I've expected you to see before-- softmax, cross-entropy loss-- basically, we have this model that we're trying to use to learn to classify something like clownfish. And we assume that we've seen some of these really basic loss functions that we use for this. So for example, if you have this as your last layer, and now you have some set of categories you're trying to learn to categorize over, the simple version of this is you take the argmax, and you say, OK, well, the thing that's darkest on here is clownfish. And then the loss that you would calculate is, OK, the actual correct ground truth label is clownfish, and then the error between those two, in this case, is zero. So it's small. So you've gotten the correct prediction.

And then maybe, actually, the real ground truth label is grizzly bear, and you've said clownfish. Now what you'd want is for your loss to be large. So you want some loss function that does a good job of punishing you when you're wrong and not punishing you when you're right.

And the way we actually do this often is if you assume the net output is some normalized vector, then you can think of this as a probability distribution over different possible object classes. And so what you want this loss function on the right, this is called cross-entropy loss. This essentially indicates the distance between what the model believes the output distribution should be and what the input distribution actually is, which in this case is what we call a one hot encoding, or essentially a vector that's 0 everywhere except for the place where it's correct, where it's 1.

And so essentially, it's just telling the network to maximize the probability of the training data, which forces the output to be a reasonably good probability model of the object class, given the image, assuming you have enough training data. So what this looks like in practice, you say clownfish, the ground truth label is clownfish. Maybe this is your prediction, so now the score is basically saying, this is how much better you could have done-- how much better you could have done, which is the difference between the amount of probability you gave the class clownfish.

I'm putting probability in quotes here. It's super important to know that these are not true probabilities. They're scores. And then that red is the dimension that you could do better. And that's telling you the direction of the gradient you want to go in.

So maybe grizzly bear, you got it correct. It's grizzly bear. Now the amount that's left over is small versus actually if it's chameleon, and you're saying iguana. Essentially, this thing is normalized, so if you put a lot more probability on one thing, then the correct answer, then you end up with a much smaller amount that now you're looking at the difference between it.

And basically, the idea with all of this in deep learning is you're taking all these different weights in your massive model, and then you're fiddling around with them until you match the desired output for all your training data. You're basically just wiggling around all these different weights.

So you're going through your training data, you're predicting something, you're calculating a loss, and you're doing this for many different data samples. And you're doing it over and over and over again until the model gets everything right, or as close to everything right as possible.

We also expect that you've seen some amount of parallel processing and the idea of tensors before this. So this is the idea that a lot of this stuff, because it's repeated calculations, you can be done in batch. So each of these individual losses for the individual data points, well, it's all going to be summed anyway. So you can do it in parallel, and now you can take these batches and more efficiently calculate that sum.

And so look, you take the same layer for all three of these, you can stack them, and now essentially you get this matrix that's features multiplied by images. And so that's what your tensors are. They're these multi-dimensional arrays, and every single layer is some representation of the input data.

And actually, everything is a tensor. A tensor is just a multi-dimensional matrix. So you can say, OK, here's our mathematical representation of this model. But you can also just represent it this way. You have these vectors of your different input values, and now you have your batch dimension, which is now this vertical stacking of all those layers.

And now you multiply that by your weights matrix, and you get out your output, and then you run that through a pointwise non-linearity, and then again, just another matrix multiplication. And then you get your output z2, and then you take that non-linearity, and you get y. So this is why the fact that GPUs can do so many multiplications in parallel was important, because we can rewrite all of these models as just essentially sets of multiplications. And we'll go into this in a lot more detail as well in the future.

So what we're going to cover in this class is how deep networks represent data. So essentially, it's this idea, where deep networks are a more compact way of representing knowledge in data. And previous methods we saw were more like lookup tables. But there's other and maybe better ways to learn a mapping between input and output that basically assume that there are these lower-level building blocks that are useful for many different possible downstream tasks.

So one nice example is the idea of trying to learn a classifier for the letter T. So if you want to learn to classify the letter T, you could try to build a classifier just for the letter T. Or you could take a classifier that classifies lines, and you could say, OK, here's a line, and then you could rotate, have a rotation of that classifier, and you'd say, here's a line. And now all you have to do is understand the connection point between these two lines. So that line classifier is reusable in some combination to get a T classifier.

And that's the basic idea here. And actually in this post hoc way, we've analyzed what is learned in layers of things like convolutional neural networks. And we do see that you'll get low-level representations in the earlier layers. Then they get combined into more and more complex representations in those later layers.

And you can actually see this as well. If you take an input image, and you take those, and you cluster them in some low-level representation of the space for each layer, the layers earlier on in the network are not going to cluster well by category, because they're really just categorizing things like, does it have directional orientations of light and dark gradients in the pixels? But actually, when you get towards the end of the network, it's learning more concepts, and that's where you start getting these clusters of different things like fish. And we will talk about actually how that representation learns and is structured through space in a bunch of different lectures on representation learning, where we talk about reconstruction, similarity, and theory.

We're also going to cover generative models going forward in the class. And this is things like text-based and image-based generation. And we'll cover basics, representations, and conditional models.

And then we're going to cover in this class how to think about weight reuse and some of the efficiencies in terms of training if you're able to reuse components of previously trained models. So here, again, this idea that, all right, well, you learned this thing to categorize pictures of animals, and now maybe you want to categorize some sort of satellite imagery, well, actually, a lot of those initial components about lines and orientations and structures, those are useful for both of these. So what do we really need to learn from scratch versus what is generally useful in terms of the representations that are learned? And this is really valuable if you don't have big data or big compute, if you can pre-generate representations that then you can learn on top of without needing to learn everything from scratch.

And so we'll get into whether or not these things are generalizable or transferable in these transfer learning lectures, where we talk about models and data.

And then finally, we'll talk about scaling. So when you think about the scale of a brain, something like this little worm only has 302 neurons, so you can think of that as 302 different values in its neural network. A fruit fly is 15,000. Human beings have 100 billion, roughly.

Elephants have 250 billion neurons in their brains. I work with elephants a lot. They're amazing.

And we'll talk about scale and deep learning and what this means, and actually get into some of the really great questions that different people asked in terms of, how does maybe scaling rules change when you're thinking about optimization, what are the laws of scaling, and how do we think about something like automatically learning how to best optimize a model?

Great. So I tried to cover today how we got where we are today, basically just honestly mostly joking, but definitely a brief history of these fluctuation and hype cycles. I would say that everyone's pretty clear that we're currently in a really hypey part of the hype cycle. What we expected you maybe saw before and also very high level, some of the things we're going to cover in this class. So I'm happy to, if anyone has any final questions-- yeah.

AUDIENCE: Slightly underweighted, but would you be willing to share an estimate for how many data it will take this first lecture to get a million views?

SARA BEERY: How long it'll take the first lecture to get how many views?

AUDIENCE: Let's say a million.

SARA BEERY: Well, I mean, honestly, the intro lecture is probably going to be the least watched of all the lectures [LAUGHS] because it's just a bunch of course logistics. We might even edit those out. But great question. I have no idea.