[SQUEAKING]

[RUSTLING]

[CLICKING]

RAMA RAMAKRISHNAN: OK, so let's get going. Today, we're going to talk about how do you actually train a neural network, because that is the heart of the game here. So just to recap, we looked, last class, at what it takes to design a neural network, and we made this very important distinction between the things that you are handed by your problem and the things that you have agency over, that you have control over.

And we noticed that the input layer for your problem, the input is the input, the output is the output. You've got to do something with the output, something that's expected. But everything that happens in the middle is actually in your hands.

And in particular, we noticed that we have to decide how many hidden layers we want; we have to decide, in each layer, how many neurons to have; and then we have to decide what activation to use, even though I'm kind of cheating when I say that because I told you very clearly on Monday that for the hidden layer activations, just go with the ReLU activation function. You don't have to think deep thoughts about this.

But the other things are all choices you have to make, and we will talk a bit later about how do you actually make those choices? OK. Now The rule of thumb, the rule of thumb always is to start with the simplest network you can think of, and if it gets the job done, stop working on it. If it's not good enough, make it slightly more complicated. So that's the meta thing you have to remember always when you're designing these things

OK, so that's what it takes to design a deep neural network. So what we will do in this class is we'll actually take a real example with real data, and then we'll think through how we would design a network to solve this problem. And while doing so, we will cover a whole bunch of conceptual foundations such as optimization, loss functions, gradient descent, and all that good stuff. All right.

So the case study, or the scenario here, is we have a data set of patients made available by the Cleveland Clinic, and essentially, we have a bunch of patients. And for all these patients, the setting is that they have come into the Cleveland Clinic, and they have not come in with a heart problem, they have come in with something else. Maybe they just came in for a physical. And we measured a whole bunch of things about them.

And the kinds of things we measured are demographic information, like what's their age, gender, whether they have any chest pain at all when they came in, blood pressure, cholesterol, sugar, so on and so forth. You get the idea. Demographic information and a bunch of biomarker information.

And then, what the Cleveland Clinic did was they actually tracked these people and figured out, in the next year, did they get diagnosed with heart disease or not? In the next year. Which means that maybe you can build a model, when someone comes in, even though they didn't come in for a chest problem, maybe you can predict that something's going to happen to them in the next year. It's a nice classic machine learning setup.

All right. So this is the thing-- so what we want to do is we can totally solve this problem using decision trees neural net-- I mean, sorry, random forest and gradient boosting and all that good stuff you folks may have already learned from machine learning, but we will try to solve it using neural networks. This is an example, of course, of what's called structured data because this is all data sitting in the columns of a spreadsheet.

So working with structured data is the way we warm up our knowledge of neural networks. And then we will do things like working with unstructured data, starting next week, with images, and then later on with text and so on and so forth. OK, any questions on this? OK. Yes?

STUDENT: Just connected even to last time's class where we took the same example, and first it was a logistic, and then we did a neural network. So the probability in case of one was 0.85, then was 0.22, and here as well. How do you know when to use what? Usually in textbooks, when to use logistic or when to use something else, but in this case, when do I complicate it to neural networks vis-a-vis, in this case, maybe just doing a random?

RAMA RAMAKRISHNAN: It's a great question. When do you use what? So I think there are two broad dimensions that you have to think about. One broad dimension is, how important is it that you need to explain or interpret what's going on inside the model to perhaps a non-technical consumer? The other dimension is, how important is sheer predictive accuracy?

In some situations, predictive accuracy trumps everything else, in which case, just go with it. In other cases, explainability becomes a big deal because if they can't understand it, they won't use it. And in those cases, it's probably better to go with simpler models, such as decision trees and neural-- I mean, not-- decision trees, maybe even random forests, certainly logistic regression, those are all a little more amenable.

But that said, even complex black box methods like neural networks, there is a whole field called mechanistic interpretability, which seeks to try to get insight into what's going on inside these big black boxes. So the story isn't over, but that's just the first-cut way you analyze the problem.

OK. So, let's get going. So if you want to design a network-- all right. So we design the network. So we have to choose the number of hidden layers and the number of neurons in each layer. Then we have to pick the right output layer. So here, what I did is-- the simplest thing you can do is, of course, just have no hidden layer. So if you have no hidden layers, what is that model called? Yes, logistic regression.

So, of course, we want to do neural networks, so I'm going to have one hidden layer because that's the simplest thing I can do. And then, I'll confess, I tried a few different numbers of neurons in this thing, and when I had 16 neurons, it actually did pretty well. So there was some trial and error that went on before I landed on the number 16.

And for some reason, people always use powers of 2, so may as well do that. So I tried 4, 8, 16, and 16 was really good. And it turns out, when I went above 16, it started to do badly. And it started to do badly because of something called overfitting, which we're going to talk about later. So, yeah, 16.

And then by default, I use ReLUs. So 16 ReLU neurons. And then here, the output is a categorical output. Heart disease, yes or no? 1 or 0. Classification problem. Which means that we want to emit a probability at the very end, therefore, we will use a sigmoid. So, so far, so good, right? Any questions? All right.

So, we're going to lay out this network visually. So we have an input. And so I just have an input. And as you will see here, x1 through x29, that's our input layer. And you may be wondering, 29? Where did he get that from? Because that doesn't seem to be 29 rows here of independent variables.

So it turns out, there are only 13 input variables here, but some of them are categorical. So what I ended up doing is to take each categorical variable and one-hot-encode it. And when you do that, you get to 39-- I'm sorry, 29. All right. And when we actually do the Colab later on, I'll show you exactly how I one-hot-encoded it, but that's what I'm doing here. That's why you have 29, not 13.

OK. Now, obviously we have decided on these hidden units, 16 units, with nice ReLUs here. And then we have an output layer with a sigmoid. And I got bored of trying to draw all these arrows, so I just gave up and said, assume there are arrows between all these things. Good? Yeah?

STUDENT: Yeah, I'm sorry, I think you already mentioned this, but why 16 units?

RAMA RAMAKRISHNAN: Why 16? I tried a bunch of different numbers of units, and at 16, the resulting model did well, so I just went with that.

STUDENT: And the logic of why it's a ReLU?

RAMA RAMAKRISHNAN: Oh, why a ReLU? Yeah. So there is just a mountain of empirical evidence that suggests that ReLU is a really good default option for using as activations in hidden layers. There is also a really great set of theoretical results, and I'll allude to some of them when we actually talk about gradient descent. Yeah?

STUDENT: So a quick question. You mentioned, in the input layer, how did you get to 29 again when you had 13 variables?

RAMA RAMAKRISHNAN: So some of those 13 variables are categorical variables, like cholesterol, low, medium, high. And so I took them and one-hot-encoded them. So if it had five levels, I would get five columns now. Yeah? And by the way, folks, just like-- yeah, just like-- please use a microphone so that people on the livestream can hear your question. Yeah, go ahead.

STUDENT: Sorry, just one question. So the vectors, since we didn't represent them, are we assuming every x is connected to all the units?

RAMA RAMAKRISHNAN: Correct.

STUDENT: And this is also a parameter that we have to decide or--

RAMA RAMAKRISHNAN: That ends up being the default. And we will see deviations from that assumption when we go to Image processing and language processing and so on, but when you're working with structured data like we are doing now, that's the default. OK, so let's keep going. So this is what we have.

So remember what I told you in the last class. Whenever you're working with these networks, get into the habit of very quickly calculating the number of parameters. Just do it a few times the first few times so that you really know cold exactly what's going on. So, yeah, how many parameters do we have here? How many weights and biases? You can work through it. You don't have to tell me the final number. You can say x times y plus z, stuff like that. Yeah?

STUDENT: 65. You have 48 weights and 17 biases.

RAMA RAMAKRISHNAN: OK, and how did you come up with that?

STUDENT: So for the weights, you have-- for the first layer, it's 2 times 16. Then for the second connection, it's 1 times 16. And then the biases, it's 16 hidden plus the outputs.

RAMA RAMAKRISHNAN: OK. Any other views on this?

STUDENT: I think it's 29 into 16.

RAMA RAMAKRISHNAN: 29-- OK, 29 into 16.

STUDENT: And then 16 into plus-- I mean 16 there.

RAMA RAMAKRISHNAN: Yeah.

STUDENT: And then biases, 16 biases and one bias.

RAMA RAMAKRISHNAN: Right. So, the way it's going to work is we have 29 things here, 16 in the middle. So 29 into 16 arrows. And then for each of these fellows, there is a bias coming in, so that's another 16. Plus, you have 16 times 1, which is here, plus there is one bias for this one. So the total is 497.

So, you can see here, there is something very interesting going on, which is that when you go from one layer to another layer, the number of weights is roughly on the order of A times B. The number of units. And so that's a dramatic explosion in the number of parameters. And that's something we have to watch for later on to prevent overfitting. OK, that's where the explosion of parameters comes from the fact that each layer is fully connected to the next layer. But we'll revisit this later on. OK.

So what I'm going to do now is I'm going to actually translate this network, the one that we have laid out graphically, into Keras code to demonstrate how easy it is. So I will give a fuller intro to Keras and TensorFlow later on, but for now, just suspend your disbelief, we will just try to do it in Keras as if we know Keras. So let's try that. Later on, we'll get into all the gory details and train it in Colab and so on and so forth. OK, all right.

So, the way we typically do it is that once we have a network like this, we typically start from the left and start defining each layer in Keras, one after the other. So we flow left to right. So let's take the input layer. The way you define an input layer in Keras is really easy. You literally say keras.Input. And then you tell Keras how many nodes you have in the input coming in. In this case, it happens to be 29, so you tell it the shape, shape equals 29.

And the reason why we say shape as opposed to length is because, as you will see later on, we don't have to just send vectors in. We can send complicated things into Keras. And those complicated objects could be matrices, it could be 3D cubes, it could be 4D tensors, and so on and so forth, so it's expecting a shape. What is the shape of this thing you're going to send me?

In this particular case, it happens to be a nice list or a vector, so it's 29. OK, that's it. So we write this down. This creates the input layer, and we give it a name. And the name here means this layer, whatever comes out of this layer has a name input. Good. Next.

You specify the shape of the input, as I mentioned, right there. Then we go to the next one. And here-- and we will unpack this, the way you define a layer is typically-- a hidden layer, keras.layers.Dense and all this stuff. So what this is is-- first of all, it says, I want a dense layer. By dense layer, I mean a layer that's going to fully connect to the prior and the later layers. Fully connect. That's what the word "dense" means.

Number two, I want 16 nodes here in this layer. Finally, I want to use a ReLU. See how compact and parsimonious it is. And that is the appeal of Keras. It's very easy to get going. So the moment you do that, you've actually defined this layer. But what you have not done is you have not told this layer what input is going to get. Because as far as this layer is concerned, it doesn't know that this other layer exists. So you need to connect them. Yes?

STUDENT: Do we need to define for the ReLU where the bends are? Like, where you take the max?

RAMA RAMAKRISHNAN: No, the ReLU, the bend is always at 0.

STUDENT: OK. Thank you.

RAMA RAMAKRISHNAN: OK. All right. So that's what we have here. And then, what we do is we have to tell it-- you want to feed this layer, the output of the previous layer. So you feed it by taking whatever is coming out of this thing, which is called input, and you basically stick it in here. So the moment you do that, boom, it's going to receive the input from the previous layer.

And because this one's output needs to go to the final layer, you need to give a name to that output. So you give it a name. I'm just calling it h4 because it's coming out of the hidden layer. It's just a variable, you can call it anything you want.

Now what we do, we go to the final output layer, and this is what we use. The output layer is just another dense layer. That's why I use the word dense. But we say, hey, give me just one thing because I just literally just need one unit here because I need to emit just one probability. And the activation I want to use is a sigmoid. Done.

And once you do that, you have to feed it the input from the second layer, so you stick an h here. Now we have connected the third and the second layers. And after you do that, you give a name to the output coming out of that, we'll just call it output. You can call it y, you can call it output, you can call it whatever you want.

So at this point, what we have done is we have mapped that picture into those three lines. That's it. But we aren't quite done yet. There's one little thing we have to do. So what we have to do is we have to formally define a model so that Keras can just work with this model object-- it can train it, it can evaluate it, it can use it for prediction and so on and so forth.

So we tell Keras, hey, create a model for me, keras.Model, and basically where the input is this thing here and the output is that thing there. And then the whole thing, we'll just call it model. So that's it. We are done. That is the whole model. That is-- it sounds really fancy, right? A neural model for heart disease prediction. That's pretty cool. Four lines.

And we will show how to train this model with real data, and so on and so forth, and use it for prediction after we switch gears and really get into some conceptual building blocks. We had a question.

STUDENT: Can you define a custom activation function that is not in the list of characters library?

RAMA RAMAKRISHNAN: Yes. Yeah. You can define-- the question was, can you define a custom activation function? You totally can. And in fact, I mean, the kind of flexibility you have here is incredible. And these innocent four lines, unfortunately, hide the potential that's possible here, but I guarantee you, in two to three weeks, you folks will be thinking in building blocks like LEGOs.

So I'm so happy when it happens. Students come to my office hours and say, I want to create a network where I have a little network going up on top, one going in the bottom, then they meet in the middle, then they fork again, they split. I'm like, unbelievable. It's fantastic. And you're going to be doing this in two weeks, I guarantee you.

STUDENT: Yeah, in the case of a multi-class classification problem, are the output nodes equal to the number of classes?

RAMA RAMAKRISHNAN: Correct. So we will come to-- so this is binary classification. And the question is, for multi-class classification, let's say you're trying to classify some input into one of 10 possibilities. We will have 10 outputs. But the way we define it is going to be using something called a softmax function, which we're going to cover on Monday. So for now, we just live with binary classification.

STUDENT: Is there a default activation method in Keras or you have to put something?

RAMA RAMAKRISHNAN: That's a good question. I believe the default might be ReLUs for hidden layers, but I'm not 100% sure. Let's double-check that, yeah.

STUDENT: Just to get a clearer understanding, when you said that beyond 16, when you tried working on those neurons, the performance worsened. That is where you are playing around with initially 2, and then maybe 4 and 6 and 8?

RAMA RAMAKRISHNAN: Exactly, right. Could you use the microphone? Yeah.

STUDENT: Do we need to define each of the hidden layer when the model gets more complex, when we have more than one layer? Or is there--

RAMA RAMAKRISHNAN: If you have 25 layers?

STUDENT: Yeah.

RAMA RAMAKRISHNAN: Yeah, yeah, yeah. So what we typically-- good question. If you have, let's say, 100 layers. Do you actually have to type in each by hand and cut and paste? No. You can actually write a little loop, which just automatically create them for you.

And so, basically what's going on is that this little output thing you see here, this variable, this output could be the result of a 1,000-layer network with all sorts of complicated transformations going on, and then finally, it pops up as this little thing called the output.

And what Keras will do is it will be like, OK, this model has this input and has this output, but boy, this output came from incredible transformations applied to the input, and Keras will process all that very easily for you, you don't have to worry about it. It's really a beautiful example of the power of abstraction. And you will see that as we go along.

OK, so, now let's switch gears and say, once you've written a model like that in Keras, how do you actually train it? Now, training is something you've been doing a lot. So for example, when you have something like linear regression where you have all these coefficients you need to estimate, you have this model, then you have a bunch of data. Then you run it through something like LLM if you use R. And what it gives you is actual values for the coefficients. 2.8, 2.9, and so on and so forth.

So the role of the data is to give you the coefficients. Or you can think of the coefficients as really a compressed version of the data. Similarly, if you do logistic regression, you have a model like that, you add some data, you run it through some estimation routine like GNN or scikit-learn or statsmodels, pick your favorite tool, then you'll come up with something like that.

So basically what's going on here is training simply means find the values of the coefficients so that the model's predictions are as close to the actual values as possible. That's it. And to find the one that is as close to the actual value as possible, a whole bunch of optimization involved. You didn't have to worry about optimization when you did regression, linear or logistic, because it's all done under the hood for you, but for neural networks, we actually get to know how it's done because it's important.

OK, so training a neural network, a deep neural network, even GPT-4, it's basically the same process as what you do for regression. You basically-- you're just a very complicated function with lots of parameters, but ultimately, you have a network with all these question marks, you add some data, you do some training, and boom, you get some numbers.

STUDENT: You may get into this, but are we determining the architecture of the network before we train it?

RAMA RAMAKRISHNAN: Yes. Because if you don't define the architecture, Keras doesn't know how to actually calculate the output given an input. And unless it knows input/output pairs, it can't do anything more with it. OK.

So the essence of training is to find the best values for the weights and biases. And the way we think of the best values is that we basically set up a little function, and this function measures the discrepancy between the actual and the predicted values. And I use the word discrepancy because the way you define discrepancy, there's incredible amounts of creativity in the field.

In fact, a lot of breakthroughs in deep learning come because people define a very clever measure of discrepancy, and then it turns out, it actually gives you all sorts of interesting behavior. That's why I use the word discrepancy as opposed to the word error because when I say error, you might be just thinking of something like predicted minus actual. That's still limiting. Prediction minus actual is too limiting, that's why you use the word discrepancy.

So we basically define a function that captures the discrepancy between the actual and the predicted values, and these functions are called loss functions in the deep learning world. And every paper that you read, you will find interesting loss functions. There are hundreds of loss functions. Enormous research creativity goes into defining these loss functions.

All right. So, these are loss functions. And so a loss function is a function that quantifies the discrepancy. So let's say the predictions are really close to the actual values. The loss would be what?

STUDENT: Close to 0.

RAMA RAMAKRISHNAN: Close to 0. Very small. And if you have a perfect model, perfect crystal ball, what would the loss be? Exactly 0. Exactly 0. So, in linear regression, the loss function we use is called sum of squared errors. We didn't call it loss function because we were not doing deep learning, just linear regression, but that's basically the loss function.

So the loss function we use must be very matched very properly with the kind of output we have. So if your output is a number like 23, you're trying to predict demand-- like a product demand for next week for a particular product, and the predicted value is 23. the actual value is 21, it's OK to do 23 minus 21, too, as a discrepancy. The error.

But for other kinds of outputs, it's not so obvious what the correct loss function is, what the correct measure of discrepancy is. And so here, for the simple case of regression, the yi-- the i here, by the way, is a superscript which stands for the i-th data point. The i-th data point.

So what I'm saying is that for the i-th data point, this is the actual value y, and this is what the model predicted. I take the difference, square it, and I square it for each point I just average all these numbers to get an average squared error-- i.e., Mean Squared Error, MSE. So this is the easiest loss function

Now, let's crank it up a notch. In the heart disease example, the heart disease-- the neural prediction model, the prediction is a number between 0 and 1 because it's coming out of the sigmoid. It's a fraction. The actual output is a 0 or 1. One of the two, it's binary. So how would we compare the discrepancy? How would we measure the discrepancy between a fraction and the numbers 0 and 1? What is a good loss function in this situation is the key question.

So let's build some intuition around this. And let's see if my little daisy chain iPad thing works. I'm doing it on iPad so that people on the livestream can see it, otherwise the blackboard is a little tough for them.

OK, so let's have a situation here-- OK. So let's say that you have a patient who comes in, and let's say they have heart disease. So for that patient, y equals 1. The true value is 1 for that patient. And now, you have this model.

And this is the predicted probability from this model. Can people see my handwriting OK? Good. I could have never been a doctor, right? So 0, 1. It's going to be between 0 and 1 because of probability. And then this is the loss we want to have. This is the loss.

So, this patient actually had heart disease, y equals 1. So let's say that the predicted probability is pretty close to 1. What do you think the loss should be?

STUDENT: Small.

STUDENT: Small.

RAMA RAMAKRISHNAN: Sorry? Close to 0, exactly. So here, if the prediction comes here, you want the loss to be-- you want the loss to be somewhere here. But if the predicted probability is pretty close to 0, even though the patient actually has heart disease, what do you want the loss to be? Really high because it's screwing up badly. So you want the loss to be somewhere here.

So basically, you want a function that's kind of like that. You want the loss function shape to be like that. High values of probability should have low losses, low values. Probability should have high losses. Yeah?

STUDENT: I understand why it has to be increasing or decreasing, but then your statement has to be--

RAMA RAMAKRISHNAN: Yeah, yeah. So it can be linear. It can certainly be linear. But basically what you want to do is, the more it makes a mistake, the more harshly you want to penalize it. So basically what you really want is something where if it basically says this person's probability is, say, the probability-- the predicted probability, say, 1 over a million, basically to 0, you want the loss to be super high so that the model is like-- it's like a huge rap on the knuckles for the model. Don't do that! That's basically what we're doing.

And I'm demonstrating that dynamic by using a very curved and steep loss function. But you can absolutely use a linear function. It's totally fine. It won't be as effective for gradient descent later on, but a bunch of technical details. Are we good with this? All right.

So, now let's look at the case where a patient does not have heart disease, y equals 0. Same setup. Predicted probability, 0, 1, loss. So for this patient, they don't have whatever-- they don't have heart disease. If the probability is close to 0, what should the loss be? Close to 0. It should be somewhere here.

And the more and more the probability gets closer and closer to 1, you want to penalize it very heavily, which means you want the loss to be somewhere here. So you basically want a loss, ideally, that's going up like that and climbing higher and higher. Are we good? OK, perfect. Because we have a perfect loss function for that.

So, just to recap, this is what we want. People with for points with y equals 1, lower prediction, predictions should have higher loss. We want something like that. And then turns out, there's a very little simple loss function which just literally just uses the logarithm, which will get the job done.

So what you do is you literally do minus log of that predicted probability. That's it. And that thing has exactly that shape. And in fact, you can see it numerically. So if the loss is 1, it's 0. If it's half, it's 1.0. And if it's 1 over 1,000, it's almost 10. If it's 1 over 10,000, it's going to be much higher, very high losses. So minus log probability, boom, done.

Similarly, this is what we want for patients for whom y equals 0. And it turns out, if you do minus log 1 minus predicted probability, it does the same thing. OK. Mathematicians, once again, saved by the logarithm.

So in summary, this is what we have. For data points with y equals 1, we have this; data points with y equals 0, we have this. But it feels a little inelegant to say, well if it's y equals 1, I want to use this; if y equals 0, I want to use that. There's an if-then thing going on here, and I don't know about you folks, but if-then really irks me mathematically because you can't do derivatives and so on very easily.

But no worries. This is MIT. We know-- we have a bag of math tricks, so what we do is, we can actually combine them both into a single expression like this. And here, the y i, again, is the i-th data point. Remember, y i is either 1 or 0 always. And this model of x i is the predicted probability.

And I've just taken the minus log, the minus, and I've just moved it here. And I've taken the minus that was here and just moved it here. That's why you see it like this.

So this one is basically, you can convince yourself what happens. The single expression will get the job done. So let's say there is a patient for whom y equals 1. What's going to happen is that when you plug in y equals 1, this becomes 0, the whole thing will collapse to 0. While here, y equals 1 just means it becomes minus log probability, which is what we want.

Conversely, if y equals 0, this whole thing is going to disappear. And this thing becomes 1 minus 0, which is just 1, and so it becomes minus log 1 minus probability, which is, again, what we want. Simple and neat, right? So in one expression, we have defined the perfect loss. No if-thens, none of that crap. Good.

So now what we do is, that was true for every data point, but we obviously have lots of data points, so we just add them all up and take the average. That's it. We average across all the data points we have. So that we get an average loss. OK. This is the binary cross entropy loss function.

STUDENT: Is there a way you can edit the loss function so that you penalize false negatives more strongly than--

RAMA RAMAKRISHNAN: Yes. You can do all of them. Great question. I'm just looking at the basic case where it's symmetric loss. You can actually penalize overestimates much more than underestimates and things like that. And if you're curious, you can just Google something called the pinball loss.

OK. Any other questions on this? So when you see this massive deep neural network built by Google for doing something or the other, if it's a binary classification problem, chances are they're using this thing. All right. So, now let's figure out how to minimize these loss functions because the name of the game is to find a way to minimize these loss functions.

So now loss functions are just a particular kind of function. So we'll first consider the general problem of minimizing some arbitrary function. And once we develop a little bit of intuition about that, we'll return to the specific task of minimizing loss functions. How's everyone doing? Yes, no, good, bad? You have a bit of a tough-to-interpret head shake.

STUDENT: It's more like I kind of lost you where you said that the loss function and the predicted probability, how were they inversely because the understanding was that the loss function is supposed to be the sum of errors, we are averaging the errors. And when you said that--

RAMA RAMAKRISHNAN: No, no, no. No, no. Sorry, sorry. Let us stop there for a second. For each point, you define the loss. That's the whole point of the game. And once you define it, you calculate it for every point and average it. So just focus on a single data point. And so now continue.

STUDENT: So now when the heart patient has, there is more probability that they know. So when there is a person who has the heart disease, you said that you want the loss function to be high. I think I'm going back to the graph.

RAMA RAMAKRISHNAN: You want the loss function to be high if I'm predicting that they basically don't have heart disease. If the prediction is close to 0, the predicted probability is close to 0, then I'm badly wrong because in reality, they do have heart disease. And that's why I want the loss to be really high.

STUDENT: OK, so effectively, loss is my way of finding out how good my model is.

RAMA RAMAKRISHNAN: Correct.

STUDENT: Instead of saying, OK--

RAMA RAMAKRISHNAN: Or rather, how bad your model is.

STUDENT: Yeah.

RAMA RAMAKRISHNAN: Right? How bad is it? That's really what the loss function is.

STUDENT: Got it.

RAMA RAMAKRISHNAN: And you want to minimize badness. That's the whole point of optimization.

STUDENT: OK.

RAMA RAMAKRISHNAN: I guess I don't have a fully-- like, similar to the point raised before, I don't have a fully clear intuition of why exactly a log function rather than something that, say, flatter for small and then really steep later.

RAMA RAMAKRISHNAN: Those are all fantastic things, you can totally do it. The reason we pick the loss-- this function because, A, is easy to work with, it has good gradients, it's well-behaved mathematically, but there are many alternatives to it. I don't want you to think that this is the only game in town or it's the only choice for us.

We have many choices. This is really-- this happens to be a very easy choice, which also happens to be empirically very effective. And I'm happy to give you pointers to other crazy loss functions which can actually do all these things, too.

OK. All right, so, minimizing a single variable function, we will warm up by looking at this little function here, which is a-- what do you call a fourth power? Quartic, right? Yeah, Thank you. Quartic. So yeah, it's a quartic function. And this is how it looks like. And you can see, there is a minimum somewhere here, between minus 1 and minus 2. Maybe minus 1.5.

So we want to minimize this function. It's obviously a toy function, a little function with one variable, but the intuition we use here is going to be exactly what we use for GPT-4, so pay attention. So, how can we go about minimizing this function? What will we do? Yeah?

STUDENT: Take the derivative and set it equal to 0?

RAMA RAMAKRISHNAN: You take the derivative. Exactly. So you take the derivative. So let's look at what the derivative does for us, but then the second part of what I said-- yeah. Second part of what I said was set it to 0, setting it to 0 becomes problematic when you have very complicated functions. It's not clear at all what's going to make them 0, unfortunately.

But the idea of derivative is, in fact, the right idea. So we can go about this, we can calculate derivative, and that actually happens with the derivative, you can convince yourself. And if you plot the derivative, it looks like that. And, as you would hope, wherever the minimum is, in fact, the derivative is crossing-- the derivative at 0 here is crossing the x-axis, in this case we can actually do that. So let's say you have the derivative, how can you use it? What is the value of a derivative? What does it tell you? Yeah?

STUDENT: You use a gradient descent algorithm?

RAMA RAMAKRISHNAN: You are 10 steps ahead of me, my friend. I just want a basic answer. Like, what good is a derivative? Like, what does it tell you? When you calculate the derivative of something at a particular point--

STUDENT: It tells you the rate of change of the function at the place you are.

RAMA RAMAKRISHNAN: Correct, exactly right. So here, what the derivative tells us is that the slope tells us a change in the function for a very small increase in w. And this just high school calculus, I'm just doing a quick refresher. So, what that means is that if the derivative is positive, what that means is that increasing w slightly will increase the function.

So if you're here, you calculate the slope is positive, it means that if you go slightly in this direction, the function is going to get higher. Similarly, if it's negative, let's say here, you calculate the slope is like this, it's negative, which means that if you increase w, if you go in this direction, it's going to decrease the function. All right.

And if it's kind of close to 0, it means that changing w slightly won't change anything. So if you're here, changing it slightly won't change anything. That's it. So what we do is this immediately suggests an algorithm for minimizing g w, which is let's start with some random point w, and then let's calculate the derivative at that point.

And once we do that, there are three possibilities. It could be positive, negative, or close to 0. And if it's positive, we know that increasing w, will increase the function, but we want to decrease the function, we want tc minimize it, which means that we should not be increasing w. We should be doing what here?

STUDENTS: Decrease.

RAMA RAMAKRISHNAN: Yes. And similarly, if it's negative, what should we do here?

STUDENT: Increase.

RAMA RAMAKRISHNAN: Exactly. So in the first case, you reduce w slightly. In the second case, you increase w slightly. And if the thing is close to 0, you just stop because there's nothing else you can do.

This is the basic intuition behind how GPT-4 was built, which is kind of shocking if you think about it. Which means that all the heavy-duty optimization stuff that people have figured out over the decades is kind of not used. This algorithm is what's being used with some flavors on top of it.

So, yeah, so back to this. And you do that. And then, if you have run out of time or compute, or if you're run out of time and so on, just stop. Otherwise, just go back to step 2 and try again. Of course, if it's close to 0, you've got to stop anyway. Yeah?

STUDENT: Is there the concern of a potentially local minimum there?

RAMA RAMAKRISHNAN: It's coming. So that's a function. It's going to find you some point where the data is kind of close to 0. So this is called gradient descent. This is gradient descent, this little algorithm. And this very PowerPointy MBA table can be collapsed into this little expression. Basically says, calculate the derivative multiplied by a small number, which we'll get to in a second, and then change the old w to-- the new w is the old w minus the little number times the gradient.

So this little one-line formula is basically gradient descent. And what you should do, just to build your intuition, is to make sure that these three possibilities here map nicely to this. Like, this thing will actually captures these three possibilities. Can you guess when gradient descent was invented? You need to have some historical fun, right?

STUDENT: The 19th century?

RAMA RAMAKRISHNAN: 19th century. Yeah, OK. Good. Very good. Excellent guess. 1847. It was invented in 1847 by Cauchy, the great mathematician. And in fact, if you're curious, you can check out the paper. I gave you give you the paper here for handy reference. OK. So 1847.

So GPT-4 is built using an algorithm invented in 1847. Which I find astonishing, frankly. This little thing is so capable. OK. So that's gradient descent. And this little number alpha is called the learning rate, and it's our way of essentially quantifying the idea of, let's not increase or decrease w massively, let's do it slightly because the gradient is only valid for small movements around your point.

If you take a big step, all bets are off. So this alpha tells you how small a step should you take. And typically, it's set to very small values, like 0.1, 0.01, and so on and so forth.

And in fact, if you read any deep learning academic papers where they have trained like a big model to do something, a lot of researchers will very quickly go to the appendix where they have described exactly what learning rates were used because the learning rate is part of the IP for how it's built. It's a lot of trial and error that goes into these learning rates. OK, so that is gradient descent.

So if we apply this algorithm to g w, our original function, we just keep on doing this thing a few times, what you will find is that if-- let's say we start with two points-- the point we randomly pick is 2.5. We set the alpha to 1. We run this algorithm, it starts here, then it goes there, it goes there, boom, boom, boom, boom, boom, and then it finally ends up here. In, like, four or five iterations, it finds the minimum.

This is obviously a very simple, well-behaved, nice little function, so you can easily optimize it. If you want, you can just go to this thing. There's a nice animation of this thing as well.

OK. So now-- all right, before we actually go to the multivariable function, I want to go to the question that you posed about local minima. Actually, you know what? I think I may have some slides on it, so, sorry, I'll come back to this. So let's actually see what-- we looked at a toy example where there was only one variable. What if you have-- what if it was GPT-3? GPT-3 has 175 billion parameters. 175 billion. And GPT-4, they haven't published it, so we don't know. It's supposed to be 8 times as much.

So, I mean, the number of parameters is massive. So basically, our loss function has billions of variables, billions of w's that we need to optimize over, minimize over. So we need to use this notion of a partial derivative. So let's take baby steps and say, OK, what if you have a two-variable function? Something like this, very simple.

So what we can do is, we can calculate the partial derivative of g with respect to each of these w's, and the partial derivative, just to quickly refresh your memories, is you take a function, you pretend that everything other than w is a constant. Then the function becomes-- a function of just one variable w, w1.

And then you just differentiate it like you do everything else, and you get something, and that is this thing here. And then you do the same thing for w2, you get this thing here. And then you just stack them up in a nice list. This is the vector of partial derivatives.

So how should we interpret this? The same way as before. Basically, for a small change in w1, keeping w2 and everything else fixed, how does the function change if we change just w1 slightly? And similarly for w2 and all the way to w1, 75 billion. It's the same thing.

So, now, when you have these functions with many variables, many w's, since we have a gradient for each one of those w's, we stack them up into a nice vector of derivatives, and this vector is called the gradient. And it's denoted using this-- anyone know what the symbol is called?

STUDENT: Laplacian?

RAMA RAMAKRISHNAN: Yeah?

STUDENT: Laplacian?

RAMA RAMAKRISHNAN: Maybe. Maybe it's a synonym, but the one I'm familiar with is nabla. Delta is the one that's upside-down triangle, but I think the upside-down triangle is called nabla, if I recall. Am I right? Thank you. He's my go-to. So yeah. So the gradient, we just call it the gradient, and it's written as this.

All right, so what we do is we simply do gradient descent on every one of the w's using its partial derivative. So in a gradient step, we update w1 using this formula, w2 using this formula. Finished. We've just generalized gradient descent to an arbitrary number of variables.

And of course, as before, this can be summarized compactly as this vector formula. We'll just do this. So what's going on here is that we have w1 or w1 minus alpha times the function g of w1 and w2, w2 minus alpha g by w2. And then all we're doing is we're just stacking them up into a vector like that. Minus alpha.

And this vector is like that. So this can be written as just this vector w, the new vector, old vector minus alpha, and the gradient is finished. And you can see, if it's GPT-3. This vector is going to be 175 billion long. Whether it's 2 or 175 billion, who cares? It's the same thing, right? OK. So yeah, so that's what we have here.

I'm really thrilled, by the way, this whole iPad business is working out. So I was a little worried about it. OK. So, if you look at two dimensions, this function-- and if you actually look at-- if you plot the function, this is w, the first w, the second w, and then this is actually the loss function, match that function, g w, and so you want to find the minimum here, and so this is how gradient descent will progress if you're starting from this point.

Or you can also look at it from up top-down into the function. And that's what this picture is, and it shows gradient descent starting from there, and working its way down from here all the way to the center.

OK, so all right, local minima. So now gradient descent will just stop near hopefully a minimum. But the problem is, it may not be a global minimum. It may not even be a minimum. So let's see what I'm talking about here. Here are some possibilities.

So let's take a simple function. Let's say this is g w, this is w. And turns out, this function actually looks like this. So you can see here-- well, this point, this point here is a local minimum. This is a local minimum. This is a local minimum. These are all lots of local minima here. And, yeah, there's a lot of local minima here, too.

So these are all places in which the derivative is going to be 0. So if you run gradient descent and it stops because the gradient is 0, you could be in any of these places. So there is no guarantee, so this in this picture happens to be maybe the global minimum because it's the lowest of the lot, But there's no guarantee you're actually going to get there.

There's not even a guarantee you're going to be in any of these places because you could literally be in this thing here where it's taking a break and then continuing on down. That, by the way, is called a saddle point. I drew it badly, but the source of coming in-- taking a break and going down again, it's called a saddle point. So gradient descent can stop at a saddle point, it can stop at some minima. There's no guarantee it's going to be global.

But it turns out, it has not mattered. So it has not mattered. And there are a whole bunch of reasons why it has not mattered. Because when you have these very complicated neural networks-- they're very complex functions, even finding a decent solution to these complicated networks is actually really good for solving the problem. You don't have to go to the best, best possible solution. And in fact, if you go to the best possible solution, you actually run the risk of overfitting. So that's one reason.

The other interesting reason-- and by the way, this is a very hot area of research, to figure out exactly-- so it's like this. Empirically, what we have seen is that not worrying about local minima, global minima, all that stuff has not hurt us because these things are amazing. GPT-4, probably they just stopped somewhere.

They probably-- it wasn't even a local minima. They're like, oh God, it's been running for six days, it's spent $2 million, let's stop. Because these are very expensive. But that's still so magical. You don't need to get to anywhere close to local minima.

But there's another interesting point, which I read about. People basically hypothesize that for you to be at a local minimum, just think about what it means. It means that you're standing at a particular point. In every direction that you look, things are just sloping upward. Everything is sloping upward. If everything is sloping upward all around you, could you be at a local minima, by definition?

But if you have a billion dimensions, what are the odds that you're going to be standing at a point where every one of those billion is going upward? The odds are really low. Chances are, some of them are going to be going up, some of them are going down, others are coming down and going the other way, it's going to be crazy. So in some sense, the best you can hope for in these very high-dimensional situations is probably a saddle point. And it turns out, it's good enough.

So for those reasons, we are content with just running gradient descent with some tweaks, which I'll get to in a second, and it just performs really admirably. Yeah?

STUDENT: How does alpha depend on how much compute you have? Like, would you set the learning rate based on that, or not really?

RAMA RAMAKRISHNAN: No, the learning rate is really-- is a measure of-- it's like this. When you're at a point where you think that the gradient is looking nice and-- if you take a step in the direction, it's going to go down. And if you further believe that it's going to keep going down in the direction for a while, then you're very confident about taking a big step.

But if you're like, I don't know, because maybe I take a little step, maybe I have to go this way. I can't go straight anymore, then you don't want to take a big step because then you have to backtrack. So those kinds of considerations go into the learning rate.

And so that's of the rough answer to your question. It's not so much determined by compute and bandwidth and things like that. But, again, it's a complicated thing because sometimes with a given amount of compute, if you have a particular kind of data, you can have very aggressive learning rates. So it tends to be a bit jumbled up, complicated, but that's the quick surface-level idea of what's going on.

OK. 9:31. By the way, folks, this lecture is probably one of the driest in the semester because I have to go through all the concepts. Once we start doing Colabs, things get a lot more lively.

OK. All right. So, now let's talk about minimizing a loss function gradient descent. So here's our little binary cross entropy loss function that we saw from before. This is what we want to minimize. So if you look at this thing, where are the variables we need to change to minimize this function? Folks, don't look at your phones. Even with laptop and iPad use, don't look at your phones.

STUDENT: Sorry, we've kind of abstracted the variable w, but just to bring it back, those are actually the weights in the neural network, right?

RAMA RAMAKRISHNAN: The weights and the biases. I'm just calling them as weights.

STUDENT: So the output of these minimization functions are going to be the actual weights in our model?

RAMA RAMAKRISHNAN: Exactly. Exactly right. The whole name of the game is to find the weights. And so for example, when you see in the press that Meta has essentially made the weights of Llama 2 or something available, that's basically what they have done. Basically published the weights.

STUDENT: The reason that's so valuable is--

RAMA RAMAKRISHNAN: Microphone, please.

STUDENT: Oh. Because if you have a billion parameters, the compute time on that is horrendous and expensive. That's why those weights are so valuable?

RAMA RAMAKRISHNAN: Correct. The weights are the crown jewel because they are the result of a lot of money and time and smartness being spent. There is a separate question of why are they making it open source? I'm happy to chat about it offline. All right. Cool.

So what are the variables we need to minimize? It's basically the parameters, and they're hiding inside the model term. Because what is the model? The model is some function like that. If you look at the simple GPA and experience thing we looked at on Monday, we finally figured out that the actual thing that comes out here is going to be this complicated function of all the x's and the w's and so on and so forth, and that complicated thing is showing up inside this thing.

And the w's here are the variables we need to change to minimize the loss function. And it's important for you tc note and understand that the values of x and y and so on are just data. You're not optimizing anything, they're just data. What you're optimizing is the w's. The weights.

OK. So you imagine replacing the model here with the mathematical expression above, whenever this appears the loss function. And once you do that, your loss function is just a good old function of the w's. The fact that it's a loss function is kind of irrelevant. It's just a function.

And since it's just a good old function of the w's, you can apply gradient descent to it as we normally would. It's no big deal. Which brings us to something called backpropagation. If you remember nothing else about backpropagation, just remember this-- never used the word backpropagation again. Only use the word backprop. You're hip and cool to the deep learning community. Backprop.

All right, so what is backprop? Backprop is a very efficient way to compute the gradient of the loss function. So when you have this loss function, and let's say you have a billion w's, and you have 10 million data points-- so, the little n we saw was 10 million. That is a lot of computation. And that is just for one step of gradient descent.

So backprop is a way-- is a very efficient and clever way to compute the gradient of the loss function, which takes advantage of the fact that what we have here is not some arbitrary model, it's a model that came from a particular kind of neural network, which has layers one after the other, and then there was an output at the very end.

So what backprop does is it organizes the computation in the form of something called a computational graph, and the book has a good discussion about it. And so what we do is we start at the very end, we calculate the gradient of the loss with respect to the output. Then we move left, we calculate the gradient of that output with respect to the output of just the prior hidden layer. Step to the left, calculate the gradient of the current with respect to the previous layer. You get the idea.

It's iterative and it moves backwards, and by doing so, you never repeat the same computation twice wastefully. That's the big advantage. You calculate once and reuse it many, many, many, many times.

The second advantage is that if you organize it this way, it just becomes a sequence of matrix multiplications. And [AUDIO OUT] sequence of matrix multiplications and eliminates redundant calculations. And, best of all, there are these things called GPUs, Graphics Processing Units, originally invented to accelerate video game rendering, and as it turns out, to accelerate video game rendering, the core math operation you do is basically matrix multiplication. Some linear algebra operations.

And so someone really, at some point, had the bright idea for deep learning, calculating gradients, and so on, we need to do matrix multiplications, and here are some specialized hardware that does really-- that does a fast job of matrix multiplications. Can we use this for that? And they did it, and all hell broke loose. That's literally what happened. And that's why NVIDIA is valued at about $1.5 trillion or something?

So, yeah. So they are really good. And so backprop, the way you do backprop, plus using it on GPUs, leads to fast calculation of loss function gradients. If this thing were not true, this class would not exist because there wouldn't be any deep learning revolution. This is a fundamental, seminal reason.

All right, so the book has a bunch of detail. And I actually did-- I hand-worked-out an example of calculating a gradient the old-fashioned way, and calculating it using backprop. So take a look at it, I'll post it on Canvas, and you will understand exactly where the savings come from, where the efficiency gains come from. Because of time, I'm not going to get into it now. All right. Any questions so far? Yep?

STUDENT: Sorry. I followed up to-- and so we've done gradient descent, which is different than calculation of the gradient of the loss function. What is the purpose of the calculation of the gradient of the loss function?

RAMA RAMAKRISHNAN: You calculate the gradient because the fundamental operation of gradient descent is to take your current value of w and modify it slightly, and the modification is old value minus learning rate times gradient. It would be cool if I say go back five slides to this thing and it just goes back. Product idea. Anyone? Startups? So this one.

So this is the fundamental step of gradient descent. So this is the current value of w. You calculate the gradient at that current value multiplied by alpha, do this thing, and you get the new value. And you keep repeating.

STUDENT: Right, but g w, that's not the loss function?

RAMA RAMAKRISHNAN: It is a loss function.

STUDENT: That is the loss function?

RAMA RAMAKRISHNAN: Yeah. Here, I'm just using g as an arbitrary function to just to demonstrate the point, but when you're optimizing-- when you're training a neural network, what you're actually doing is minimizing a loss function, loss of w.

STUDENT: Sorry, I got things mixed up.

RAMA RAMAKRISHNAN: No worries. Yeah?

STUDENT: How do we define the initial weights for the neural network?

RAMA RAMAKRISHNAN: So, yeah, the initial weights-- so there are many ways to-- so first of all, they are initialized randomly, but randomly doesn't mean you can just pick any random weight. There are actually some good ways to randomly pick the weights, and those are called initialization schemes. And there are a bunch of very effective initialization schemes people have figured out over the years, and those things are baked into Keras as the default.

So Keras, I believe, uses something called the HE initialization, H-E initialization, or the Xavier Glorot initialization. I wouldn't worry about it, just go with the default initialization. And the reason why they have to be very careful about how these weights are initialized is because if you have a very big network and if you initialize badly, then the gradient will just explode as you calculate it.

The earlier layers, the weights will have massive gradients or gradients will vanish, so therefore, call the exploding gradient problem or the vanishing gradient problem. To avoid all those things, researchers figured out some clever way to initialize so that it's well-behaved throughout. Yeah?

STUDENT: If using backprops and GPUs was so critical, I'm just curious, who first did it and when? Was this a couple years ago? Was it a company? Was it--

RAMA RAMAKRISHNAN: Yeah. Well, GPUs have been used for deep learning, I want to say-- I think the first case may have been in the mid-2005, 2006 sort of thing. But I would say that it burst out onto the world stage and made everyone take notice when a deep learning model called AlexNet in 2012 won a very famous computer vision competition.

And it beat the-- and it set a world record for how good it was, and that's when everyone was like, hey, what is this thing? And that's really when it burst onto the world stage. I'll talk a bit more about it when I get into the computer vision segment of the class. But you can Google AlexNet and you'll find a whole bunch of history around it.

STUDENT: Hypothetically, then, is it true that if we could get to a global minima, that would mean there would be no hallucinations?

RAMA RAMAKRISHNAN: Aha, good question. So if it is perfect, if you get to a global minimum-- first of all, global minima doesn't mean the model is perfect. It may still have some loss. But global minima is going to be on the training data. You can imagine that the test data, future data has its own loss function. So what is minimum here may not be minimum there. That's the problem. Is it a comment? No? OK.

STUDENT: Just saying that that would mean that also you can be overfitting--

RAMA RAMAKRISHNAN: Correct, exactly, exactly. So if you overdo, if you find the best thing in the training function, chances are, it doesn't match the best thing of the test data. So on the test data, you're actually doing badly. OK. So, come back to this.

OK. Now the final twist to the tale here, we're going to go from something, gradient descent, to something called stochastic gradient descent. And stochastic gradient descent, or SGD, is the workhorse for all deep learning. And funnily enough, SGD is simpler than GD. Just when you thought it couldn't get simpler. OK.

So for large data sets, computing the gradient of the loss function can be very expensive, needless to say, because it has to be done at every step, and the cardinality of the data set is really big, and you may have billions of parameters, it's just very, very tough to compute it even with backprop.

So the solution is, at each iteration-- when I say iteration, I'm talking about this step of gradient descent. Instead of using all the data, instead of calculating the loss function by averaging the loss across all n data points, and then calculating the gradient of that thing, what you do is you just choose a small sample randomly. You choose just a few of the n observations, and we call it a batch.

So for example, the number of data points-- you may have 10 billion data points, but in every iteration, you may literally grab just 32 or 64, something really small. Like absurdly small. And then you pretend that, OK, that's all the data I have, you calculate the loss, find the gradient, and just use that here instead.

So this is called stochastic gradient descent. So strictly speaking, theoretically, SGD uses just one data point, but in practice, we use what's called a minibatch. 32, 64, whatever. And so minibatch gradient descent is just loosely called stochastic gradient descent, SGD.

So an SGD, as it turns out, you can see, it's clearly very efficient because it's just processing a few at a time. And in fact, if you have a lot of data, and you calculate the gradient of the loss function, it may not even fit into memory. It's really problematic. But with SGD, it says, I don't care whether you have a billion data points or a trillion data points, just give me 32 at a time. And you just keep on doing it.

It turns out, because not all the points are used in the calculation, this only approximates the true gradient. It's only an approximation. It's not the real thing, it's only an approximation, but it works extremely well in practice. Extremely well in practice. And there's a whole bunch of research that goes into why is it so effective. And people are discovering interesting things about SGD, but we don't have a definitive theory as to why it's so good yet. We have some interesting research threads that have happened.

And, very tantalizingly, very tantalizingly, because it's only an approximation of the true gradient, SGD can actually escape local minima. So in the true loss function, you're at a local minimum, but in SGD loss function, when you're doing SGD, you're reaching the minimum of the SGD loss function, which actually may not be the actual loss function.

So as you're moving around, you're actually jumping from local minima to local minima of the actual loss function. I know, that's a mouthful, I'm happy to tell you more. It's just a side thing that I just want you to be aware of. One of the reasons why SGD is actually effective, it's almost like you work less and you do better. How many times does it happen in life? This is one of them.

Now SGD comes in many flavors. Many siblings, it's got a lot of siblings and variations. It's a big family. And we're going to use a particular flavor called Adam as our default in this course, and I'll get back to it when you get into the Colabs and things like that.

All right. By the way you know how all these pictures I've been showing you a nice little function like that, a little bowl and so on? This is a visualization of an actual neural network loss function. You can see the hills and valleys and the cracks and so on and so forth. And you can check out that paper to get more insight into how they actually came up with this visualization. It's crazy. It's complicated. Yeah?

STUDENT: So for SGD, do you perform the iterations until you minimize the loss function for each minibatch and then move to another minibatch?

RAMA RAMAKRISHNAN: Yeah. So what you do is you take each minibatch, and then you calculate the loss for that minibatch, you find the gradient. And use the gradient and update the w. Then you pick up the next minibatch.

STUDENT: So you don't pick a minibatch and try to perform the iterations on that minibatch until you--

RAMA RAMAKRISHNAN: Correct. Each minibatch, one iteration. Each minibatch, one iteration. Because if you do a lot of iterations on one minibatch, first of all, you never be sure that you're going to find any optimal solution because you're not guaranteed of any global minima.

And secondly, it's much better for you to get new information constantly because what you can do is you can revisit the minibatch later on. And that gets us to these things called epochs and batch size and so on, which we'll get into a lot of gory detail when we do the Colab. So let's revisit that question. It's a good question. Yeah?

STUDENT: When you do the backprop process, it's--

RAMA RAMAKRISHNAN: Very good. Backprop, no backpropagation. Nice.

STUDENT: I made sure.

RAMA RAMAKRISHNAN: Yes. It sounded like you started from the layers that were closest to the output and you went backward. And my question is, are you doing that once or is it looping multiple times? And then--

RAMA RAMAKRISHNAN: You just do it once.

STUDENT: Just once?

RAMA RAMAKRISHNAN: Yeah. So for each gradient calculation, you do it once.

STUDENT: Why does it want to start from the layer that's closest-- or why do you want to start it from the layer that's closest to the output?

RAMA RAMAKRISHNAN: Yeah. So basically what happens is, let's say that, just for argument, that you go in the reverse direction, you will discover that a lot of paths to go from the left to the right will end up calculating certain intermediate quantities, including the very final gradient item, again and again and again. The same thing is going to be calculated again and again and again.

So by starting from the end and working backwards, you just reuse stuff you've already calculated. So that is the rough idea. But if you see my PDF, I've actually worked out the example, and that will demonstrate what I'm talking about.

By the way, the backprop is just a-- in calculus, we have something called the chain rule. To calculate the derivative of a complicated function, you calculate the derivative of the outer function, then the inner function, and so on and so forth. The backdrop is essentially a way to organize the chain rule to work with the neural network, layer-by-layer architecture, that's all. Yeah?

STUDENT: So is it fair to say that once we are finding the local minimum, we are not optimizing to all the g w's because this local minimum is coming like from different curves from different lines, so is that fair to say--

RAMA RAMAKRISHNAN: When we are using stochastic gradient descent, yes. So in stochastic gradient descent, when you take, say, 32 data points from a million and you're calculating the loss for that 32 data points, you're basically trying to do a gradient step. The w equals w minus alpha gradient thing. You're doing it for that 32 points loss function, which is not the 1 million points loss function. That's why it's approximate.

But the approximation, instead of hurting you, actually helps you because it helps you escape the local minima of the global loss function. So it's an interesting and somewhat technically subtle point, which is why I'm not getting into it too much, but I'm happy to give pointers if people are interested. Yeah?

STUDENT: When you say you initialize the weights, you initialize for the whole network or just the end layer and then go backwards--

RAMA RAMAKRISHNAN: So usually it's everything in one shot. Because if you don't initialize everything in one shot, what's going to happen is that you can't do the forward computation to find the prediction. And so they are done independently, and the initialization schemes will take into account, OK, I'm initializing the weights between a layer which has 10 nodes on one side and 32 on the other side, and the 10 and the 32 actually play a role in how you initialize.

OK. So, the summary of the overall training flow is that you have an input. It goes through a bunch of layers. You come up with a prediction. You compare it to the true values. And these two things go into the loss function calculation, you get a loss number. And you do it for, say, 10 points or 32 points or a million points. And this loss thing goes into the optimizer, which calculates the gradient.

And once it calculates the gradient, it updates the weights of every layer using the w equals w minus alpha times gradient formula, gradient descent formula. And then you keep doing this again and again and again. This is the overall flow. This is how our little network is going to get built for heart disease prediction, this is how GPT was built. And this is how AlphaFold was built. And AlphaGo was built. You get the idea.

I mean, it's astonishing, frankly. If you're not getting goosebumps at the thought that this simple thing can do all these complicated things, we really need to talk offline.

STUDENT: There was a hand raised here.

RAMA RAMAKRISHNAN: Yeah.

STUDENT: Sorry. Just quickly, this is for each minibatch, right? so my question is, if you came with different weights for each minibatch, how do you add it up? Like, OK, this weight is the perfect combination for this minibatch, but you have way different weight for another minibatch, how do you combine those two?

RAMA RAMAKRISHNAN: No. At each point, what you do is you find you find-- you start with a weight, you run it through for a minibatch, you come up with a loss function, you calculate the gradient. And now using the gradient, you updated the weight, now you have a new set of weights, which is the updated weights. Call it w2 instead of w1.

Now w2 is using your network, and when you take the next minibatch, it's going to use w2 to calculate the prediction. And this whole flow will become a lot clearer when we do the Colabs. OK, so we have three minutes. I don't want to go into regularization overfitting in three minutes, so let's have some more questions. Yeah?

STUDENT: Can you use any activation function as long as it gives positive values for x squared or mod x or something?

RAMA RAMAKRISHNAN: You can use a variety of activation functions, but-- there's a whole literature on the pros and cons of various activation functions that you could use, but in general, you have to make sure of a couple of things.

One is that when you do backprop, the gradient is going to flow through the activation function in the reverse direction, and the activation function should actually make sure the gradient doesn't get squished. It shouldn't get squished, it shouldn't get exploded.

So those are some considerations-- these are technical considerations, but all those conditions have to be taken into account. If you can take those into account, then you're OK. That's the key thing to keep in mind.

And that's, in fact, why the ReLU is actually very popular, because as long as the value is positive, the gradient of the ReLU is just 1 because if you look at something-- oops. Because it's frozen. I jinxed it. So, sorry, livestream. If you have something like this, the ReLU is like that.

So the gradient here is always going to be 1, which means that as long as the value is positive, whatever gradient comes in like this, it just gets multiplied by 1 and gets pushed out the other side. So it doesn't get harmed or squished or anything like that. So that's one reason why the ReLU is very popular, because it preserves the gradient while injecting almost the minimum amount of non-linearity to do interesting things. Yeah?

STUDENT: If you have a high number of dimensions, can you do minibatching on features, dimensions instead of just observations? Keep the same number of observations, but just take a small sample of the number of features that you're actually using?

RAMA RAMAKRISHNAN: Oh, I see, I see. So you're saying-- let's say you have 10 features. Instead of taking all data points or 10 features, what if you choose five features and just use them and do this thing? As long as you can actually compute the prediction. To compute the prediction, you may need all 10 features. Or you need to have some defaults for those features.

But if you define defaults for those other five features, you're basically using all features. So that's the key thing. Can you actually calculate the prediction by manipulating? And typically you can't. All right. OK, folks. 9:55, I'm done. Have a great rest of your week, I'll see you on Monday.