# Module 15: Gradient Descent and Optimization

## Video Transcripts

### Video 1: Minimizing an Arbitrary Function Using Guess and Check

Hello again everybody. Today we're going to be talking about gradient descent. This is a technique that's used to minimize functions. And it's the technique that operates under the hood for all of the algorithms we've talked about in our course. So, we're going to start by just looking at not something in the machine learning context, but just the contexts of functions in general.

So, what I'm going to do here after importing some libraries, is I'm going to define this arbitrary function here of x. My arbitrary function is x^4 minus 15x^2 + 80x^2 minus 180x + 144/10. Why? Okay, well, I'll show you why. When we plot this function, you'll see that it has a nice shape. So, this function, it's a...it's a fourth-order function that has two local minima. And what we're going to be trying to do is find the x which minimizes this function. Which you and I can see immediately, is somewhere around 5.36. But there's also this local minimum here. So, this simple function is going to be a nice playground for us as we try and understand what will ultimately be gradient descent.

Now zooming out just a bit, thinking about how this relates to machine learning. Recall that we're often trying to find parameters which find the minima of loss surfaces. And so, this is a stand-in for the moment, for a

loss surface we might have for, say, linear or logistic regression. So, we discussed in an earlier module the existence of a function called scipy.optimize.minimize. And the way that it works, as a reminder, is that it takes a function and a starting guess, and it tries to find the minimum. So if I say, scipy.optimize, please minimize this arbitrary function starting at 6 as my starting guess, it ends up at 5.326. So, in other words, we say start here and it's able to find the bottom at 5.326. Okay?

Now there is some other information here, which I'm not going to talk about it all today. But it's also provided, and this gives some troubleshooting information if things go wrong. One that I will mention, is this is the minimizing value. So, if we plug in that value of 5.326, we get negative 0.69. OK. Now, as it happens, the scipy.optimize.minimize function. The start point you give it does actually matter. So if, for example, we start on the left side—so over here at one—you'll see that it's actually going to get stuck here, at this other lower minimum. OK? So if we run this here, we end up with x is 2.392. OK. And so this minimum, it is a local minimum, but it's not the global minimum. And this gives us some insight into what gradient descent is because it turns out that scipy.optimize is using a version of gradient descent. It is going down the slope and trying to find the bottom point. Okay?

So that's just an intuitive picture and we're going to work ourselves towards the algorithm gradient descent. But before we do that, let's try and see how we might minimize it, just totally from scratch. So, if I just gave you this function, and I said, give me the minimum. How might you do it? Well, one approach is, we might do guess and check. So we could say, okay, let's try 6. Well 0, maybe what happens if I do 7? Oh, it's a bigger number. Okay,

that's not so good. Maybe I try 3. Okay, now it is 0 again. How about 5? 0h, it is negative 0.6. What about 4.8? And I can keep doing that over and over until I get the smallest number possible. Eventually, maybe I get lucky, and I find the minimum somewhere around 5.3. So, that's one approach. It is just manually guessing and checking. That is of course not what gradient descent does, or what scipy.optimize.minimize does. But it's something we could do.

Another approach is, we could create a function called simple_minimize. And all it's going to do, is try out a bunch of different x values and then tell me the smallest one. Alright, so this is just saying, compute a bunch of y-values and then give me the argument that results in the smallest y-value. Now some of this code might seem slightly unfamiliar, but it's relatively simple Python code that you can look up what, for example, argument does online. Now the important point here is just to say what does. It just tries a bunch of different values, and it gives you the smallest that it happens to try out.

So here this arbitrary function, we're plugging in the values between 1 and 7, and we're trying 20 different such values. In other words, the guesses that I'm trying here, are the following values: 1, 1.31, and so forth. So, this is basically just the manual guess-and-check approach we did earlier, except we've automated the procedure of trying out a bunch of values. And so, the result of this function is 5.42. That's the best value that it came up with. And you'll notice that's fairly close to the true minimum.

Now, as another way to think about what just happened, we can actually do a plot. So, I already have filled out here some code that just plots the

function. And so, if I do that, I'm plotting in blue the function as before, and then in red I'm plotting all the different guesses one might have. So before, if I want to match what we just did, if I say 20 values between 1 and 7, it's basically just computing all those red dots and then telling me which one was the smallest, which happens to be this one right here. And I can increase the resolution of this, of course, by just saying, now I want to try 200 different guesses. Then it'll give me the best among all of these red dots.

So, this first approach, which will be the end of this video. It's, it works, but it has several major flaws we'll be addressing next. So, one of the problems is, if the minimum is not in this range, let's say the minimum is actually over here at eight or nine or negative two or something, our answer's going to be completely wrong. Now, even if our range of guesses is correct, if the guesses are too course—so for example, let's suppose we're only doing five such values—then our answer might not be very correct. In this case, it's pretty close, but you might not get that lucky.

And then third is that, well, if our answer, or if...even if we do get an answer that's pretty good on a really high dimensional surface—let's say we're trying to optimize maybe hundreds of parameters at once—this is going to be so computationally inefficient that it's going to be useless when we're trying to, say, do logistic regression on 100 parameters. And so, this vast number of guesses means this procedure is generally useless. But at least it's a starting point and we'll be building on that as we move forward.

## Video 2: Visualizing the Derivative

The trouble with our first approach is that we had to come up with all of our guesses in advance. And we don't really know, when we start, what the right guesses are. Otherwise, we would be in great shape. So, an alternate approach is to use the derivative of the function we're trying to optimize as a guide. Or, as we'll see when we get to higher dimensions, we'll talk about the gradient of the function. But we'll just start with derivatives of one-dimensional functions.

So, the idea is this: If the derivative of the function is negative, as you see here, then that means the function is decreasing. So, we should go to the right. And, if the derivative of the function is positive, that means the function is increasing. So, we should go to the left.

Let's see that visually to make that more clear. Okay. So here, what I've got is our arbitrary function in blue. And in red I have the derivative, the instantaneous derivative, of the function. So, as I move this around, we're looking at the slope of the function. Right. So, if we are at a guess of 1.4, the derivative is negative, that is, the curve is decreasing. And so, that means if we're trying to find the minimum, we should move to the right. We should go into the positive x-direction. By contrast, if we were, say somewhere over here, like the guess 6, the slope is now positive, which means we should pick a smaller x if we want to work our way towards the minimum. Okay. So, that right there, is the insight that we're going to use in order to optimize our function.

Now, I can also show you another way of thinking about it. This is going to be the entire derivative of our arbitrary function. So, here I'm just using our usual plotting tricks. In blue I have the function we're trying to optimize. In red dotted lines we have the entire derivative of the function. And in green I'm highlighting just 0, so we can see where the zero crossings are. Okay. So, if we look at the derivative of the function here, we can see that there's three very special places where the derivative crosses 0. In other words, it's places where the blue curve is flat.

So, when the blue curve is flat, the red dotted curve is 0 because it is the derivative of the blue curve. And so, it's at each of those points where we have a crossing of 0, that we can have minima or maxima of our functions. And here I'm assuming you've seen this in calculus. But if you haven't, I think it makes some intuitive sense. The bottom or the tops of our functions will be where the derivative is 0, in other words, where it's flat. Okay. So, what we're going to try and do then is we are going to use gradient descent to go down the curve—or derivative descent I might call it—until we reach a point where the derivative is 0. And at that point we'd be done.

Okay. Here's another way of looking at it. One more. So, here I have a bunch of code that's just going to plot our function and its instantaneous derivatives in various locations. This is basically just the same that we saw in this demo, only now in our notebook. So, this, if you want to play around with this, I will provide this notebook. You can look at the function at a particular point. So, if I want to know what is the function at position 3 and what is its tangent, this code will show it to you. Then you might find this to be a useful guide for visualizing what our procedure is going to be doing. Namely, if the derivative is positive, we're going to be picking a smaller x.

## Video 3: Manually Descending the Gradient

OK, so now that we have this idea that the derivative will inform our next guess, let's just try and do this in code. It's more straightforward than you might expect. So, what I'm going to do here is, I've set up a print statement where I'm going to start from some guess. And I'm going to print out my guess, the function at my guess, and the derivative of my function at my guess. What is the derivative of this function? How did I get that? Well, if you scroll back up at my code, you'll see that I wrote it out here. The derivative of the arbitrary function is this right here. And I computed this using the rules of calculus. And so, if you've studied calculus before, it should be very intuitive. Otherwise, I'll leave it as something that you can look up later.

How do you compute the derivative of a function? OK. So, what I'm going to do then is at any given point, I'm given an x, an f(x), and the derivative f'(x). And so, when I run this, as you see, I get, well, at 4 the value is 0 and the derivative is negative 0.4. And in fact, we can visualize that by changing this value up here to 4. You can see the slope is slightly negative. And so, here's where the next guest comes in. I want to increase x slightly, right? I know that the minimum, or I expect that the minimum, is going to be over this way a little bit. So, I'm going to say, my next guess is 4 + 0.4. The derivative is giving me a sense of which way to go. And here's the other insight: It's also giving me a sense of how much to go. Because if the derivative is very steep, well, the idea is presumably I should go further because I'm very far away from a minimum.

So, I'm going to use 4 + 0.4 as my next guess. And the answers aren't here, but I'll run it again. And I get, OK. At this point, f(x) is smaller, and the derivative is actually even slightly steeper. So, if I go back up here and I change this to 4.4. Now you will see that the slope is a little bit steeper. OK. Now I've actually written yet one more function we can use to make this process even a little more obvious to follow, instead of having to type some numbers down here and then go back up here to visualize. And that's this function I'm going to call plot one step.

So, the plot one step is going to do just what we said. It's going to take a guess. It's going to compute the f(x), the derivative of f'(x). And then using that, it's going to compute the next guess and then plot that next guess. OK. So in other words, if here I do plot_one_step(4), my old guess was 4 and my new guess was 4.4. OK, so this is where I was, and now I've moved here. Now, next, I can say, OK, let's try another guess. So, here I'm going to click. Oops, alright, here we go. Plot that again. Like I said, I do plot_one_step(4.4). And now my next guess is 5.04. Right. So, I've moved from here to here. Notice the jump was actually bigger. We went from 4 to 4.4 because the slope was 0.4. When we got to 4.4, the slope was bigger, it was 0.64, if you recall from above. Right. So, the next jump was even bigger.

OK so we're getting closer to our minimum. And so then next, let's do plot_one_step and do 5.04. And so, what we're doing right now is manually descending the gradient. Now we get down to 5.4967. So, we move from here to here. That's my next step. OK. So, we'll do one more. I'll tell you what plot_one_step_better is in a moment. I know you're excited, but we'll get there in moment. So, 5.4967. The next guess I get is 5.08. OK. Now, notice something that just happened. We went from 5.0464 to 5.967 to 5.08. So,

now we've crossed back past the minimum again. So, notice the pattern. We went down and then we missed the minimum. We jumped past it. And now, in this next step, we've jumped past it again.

I guess let's do one more. Let's do one more and get a sense of what happens next. So, now I'm going to put in this whole number—all the way to the most digits. And we'll see that we end up at 4...5.4899. OK. So, we're almost right back where we started. And actually, if you plug in this whole value of...I guess, I. Sorry, I just got to do one more. It's too tempting. OK, so let's do this value where we just ended up, and you'll see that we end up very close to where we just were, back at 5.09. And so, what's happening is, we're constantly overshooting the minimum. Over and over and over.

So, this brings me to the last tweak that we're going to do to this procedure. Which is that, rather than adding the derivative, we're going to add the derivative times a small constant. OK? So, in this case, I chose 0.3. So, the only difference is that when picking our new guess, instead of subtracting the derivative, I'm going to subtract the derivative times a small number. And the intuition is, well, I just don't like this procedure where I'm bouncing back and forth. We keep missing the minimum. We're overshooting. Right. It's like fishtailing in a car where you go out of control, you turn the wheel to the left to fix it. Oh wait, now I turned it too much. Now I turn the wheel too much to the right and so forth.

So, to avoid this oscillatory behavior, I'm going to use this alternate approach. So in this one, when I say plot_one_step_better(4), you'll notice I don't move as far because I'm only moving one third as far. The derivative was 0.4, but we're only doing a third of 0.4. OK, so what happens if we go to

our next guess, 4.12? OK, now we get 4.267. OK, what if we go again? OK, 4.267, we get 4.44. We do 4...I'm actually just going to plug it in in place. Again, to keep scrolling. OK, so I do this, we get 4.64. So, we're getting a slower convergence, right? We didn't get to 4.64 until after several guesses. But the nice thing is, we're not going to overshoot with the smaller number, 4.84. And eventually we will get there. We'll keep traveling together, friends.

OK, we're getting closer. And so, we can repeat this process over and over. And I'll leave it as an exercise for you if you want to do it a dozen times or 20 times, but you will eventually get to a nice minimum. So, for example, once I get to 3.23, it'll be 3.25 and so forth. And so, we converge down to something useful. Now, the natural question you should ask yourself is: Is there a nice way to write this code that repeats this process over and over? And indeed, there is, and that'll be our next video.

## Video 4: Gradient Descent in Code

We can write out the procedure we just carried out manually as a recurrence relation. So, recall that the procedure we were just doing is—we're plugging in values over and over, painstakingly producing guess after guess after guess. And so, this equation captures that idea. Our next guess was the old guess, minus 0.3 times the derivative of the function. That's all we were doing. Now, it turns out that this little equation right here, is gradient descent. This is the magical thing that tools like SciPy optimize use in order to minimize functions.

So, one way to think about it is that given a current x, like a guess x, we create our next guess based on the sign and the magnitude of the derivative. Now in my case, I picked a value of 0.3 totally arbitrarily. We can

actually pick other values, like 0.7, or 2, or whatever. And so, we can generalize this equation by replacing that learning rate with α. So, this learning rate, α. The reason I call it a learning rate is, it captures how quickly gradient descent works towards, or learns, the minimum. And so, a large α, like α equal to one, it moves quickly, but it can overshoot. But with a small α, you move more slowly, but there's less chance of overshoot.

OK. So, given all that, what I want to do now that we have all of this groundwork, is we're going to write gradient descent. And we'll find that it's actually only a few lines of code. But those few lines of code are incredibly powerful. So, what I've got here is, I'm going to live code this up just to emphasize just how simple this procedure is. This gradient_descent function takes the derivative of the function we want to optimize. It takes the initial_guess, takes a learning rate, and a number of steps. OK, and it's going to return a NumPy array of all guesses over time.

So, how does this work? Well, I'm deleting pass, because that means do nothing. And so, instead I'm going to start by creating my guesses array, which is just initially going to be only the initial guess. That's all I've done so far. And then I'm going to create a variable called initial_guess. So, this represents the x's. OK? Now, so long as the number of guesses is less than whatever the function user gave me. So, say they run, say run it for 20 steps, as long as we've done less than 20 steps, we'll do the following. We'll say the next guess, right here, is going to be the old guess. Right. So, the current guess is equal to the old current guess, minus α times the derivative at the current guess. OK? That is the step we were doing before, where we kept plugging in a number over and over. And then, because I want to keep track of all the guesses, we'll append the current guess. We're just going to repeat

that process over and over. And then when we're done, we're going to return the guesses as a NumPy array because that's what the comment says to do. OK.

So, just these lines of code implement gradient descent and give us what I think of as the godlike power of the SciPy optimize library. Now, there are some differences. Remember that scipy.optimize takes the original function, whereas we're asking for the derivative of the function. It also doesn't require us to specify the number of the steps to do. It just runs until it finds something it's happy with. But fundamentally, this is what scipy.optimize is all about and it's really not much code. OK.

So, let's see this code in action and also make sure that my code's working properly. So, if I run this gradient_descent function that I just wrote—where I give it the derivative of our function, a starting guess, an α, and a number of guesses—it should produce a trajectory that is basically what we got when we ran this example earlier by hand. And so, indeed we run this. We see that we get 4, then 4.12, then 4.267 and so forth, up to 5.326. Right. So, these are the same numbers we got up here. And so that means that I feel pretty good that my procedure is working well.

Now, we can also try other, for example, guesses or learning rates. And so, I encourage you to try this out on your own, but I'll give you an example real quick. Here. If I pick a learning rate of say 1, we can see that it goes 4, 4.4, 5.0464. You may remember these numbers from earlier. And then it starts bouncing back and forth between 5.49 and 5.08 or so. Somewhere in that neighborhood, slightly bigger. Right. So, it's bouncing. If I go up to, say, a learning rate of 1.5 and it behaves even weirder. If I pick a learning rate of

0.1, it converges more slowly. So, that's something as I mentioned, you'll explore, but on the homework.

Another thing we can do is visualize the trajectory taken by the algorithm. So, here is going to be a function that is just going to plot where we go over time. So, if I do 4, 0.3, 20, our trajectory is we start here, and then we smoothly make our way down to the bottom. If I do 1 here instead, I see that I go down and then I'm bouncing back and forth. That's the way to interpret this. If I start at, say, 1.5, my trajectory is I do some big jump over to here and then bounce my way back and forth. We can try different values over here: 0.8, maybe 0.5. There we go. So, if we do 0.5, the learning rate, it ends up sticking us in this minimum here. And so, depending on where you start and your learning rate, you'll end up in different locations. And so, these choices do actually matter. But, in principle, this algorithm can find minima. OK. And so, here I've just run the algorithm some, some fixed number of times. And depending on the choices I make, I end up in some location. And you'll explore the behavior of this on the homework.

Now, I should note that more sophisticated implementations, like the SciPy optimize library, they do things a little fancier. Where they'll look at the...how much the function is decreasing at each step. And if that is small enough, it'll say, well, I'm basically done. Right. Things like that. And so, we're not going to discuss those ideas in our course because I only want to focus on the big picture. Because it'll be important for us to understand how machine learning algorithms are trained. So, at this point, we've now understood how to optimize one-dimensional functions using a gradient descent procedure, which we wrote ourselves. And so, next we'll go back to the world of

machine learning and understand how this can help train, for example, a logistic or a linear regression model.

## Video 5: Applying Techniques to Optimizing Linear Regression

So, let's use our gradient descent algorithm on a machine learning problem. Specifically, we're going to do linear regression on the Tips dataset. We did this a long time ago, but I'm going to repeat this because it gives us a nice test platform for gradient descent. So, I'm going to first load in my tips dataset. And then I'm going to create a scatterplot of the data. Now here I'm using the Seaborn scatterplot library just for variety. We could also use Plotly of course, but here's Seaborn. And so, as you may recall, we want to build a model where we want to predict the tip based on the total_bill.

Now, I'm going to build a model which is very simple. We're only going to have the parameter θ_1, which is going to be the slope. And we're going to have no offset. OK. And that is, there's not going to be a y-intercept. OK. So, this is going to be things we've seen before, but now with a different punchline with gradient descent. So, as I noted, we want a one-parameter model, where our output is simply going to be θ_1 times x, where x is the total_bill. And so here, if I plot a scatterplot of my data and I plot over it, y_hat equal to 0.1 times x. This red line is my guess. And so, you know, we could pick different values of this θ: 1, 0.2 goes up there, 0.15, and so forth. OK.

So, I want to find the best slope of this line. Now we already know how to do that, right? We can do that by creating a linear model, a linear regression

model in scikit-learn and then asking for its coefficients. And so, let's do that. So, if I say, create a LinearRegression model with no intercept, where I fit the total_bill and tips, I get back 0.1437. So, the best slope here is 14.37%. So, where does that come from? Well, we've actually already talked about this to some degree, because we said this is the tip percentage, which minimizes the mean squared error. I'm going to repeat some of what we did before, so that we could ultimately apply gradient descent. OK.

So, where did this come from? Well, it's the minimum of some arbitrary function, just like before. Except that in this case, instead of our arbitrary function being x^4 plus du-du-du, it's going to be a function that is dependent on this data. Every single one of these data points will affect this function. OK. So, if we use the L2 loss as our loss function, then we're going to be minimizing the mean squared error. And so, let's write a function—or I already have one written here to avoid repeating things you've already seen before. It computes the mean squared error for a particular choice of θ_1 on our dataset.

So, in particular, you give it the theta1 choice, the x's—so that's all a total_bills—and then the observed actual tips. Then I will produce my guesses of the tips. And then we'll return the mean squared error. And I could have also used the scikit-learn mean squared error function. But I've written out the whole expression here to be really explicit about what's going on. Right? So yes, I could do sklearn.metrics.mean squared error, but here I'm doing it all on my own. OK, so once I've defined this function, I can then plug in some values. So, if x are my total_bill and y are my tips, then the loss of a 10% tip is 2.077. That's the average squared error on all the tables. So, it's a function of all of these blue dots.

Now, if I change this to 0.2, I get a different loss, a larger loss. If I take 0.3, it's even worse. And we did a guess-and-check procedure in a much earlier module, but the best one would be 0.1437. OK. So, how do I find the best value? Well, let's go through that. So, recall, because this data has already been collected, it's not changing, like we already have the whole tips dataset. The only real variable is θ_1. Alright, so I'm...I can change these, but these don't change in between runs. So, it's a little awkward to be working with this mse_loss function.

So, what I'm going to do that's going to make life easier, is we're going to define a single-argument version of the mean squared error loss, where these are hard-coded values. So, this mean squared error single_arg function is just going to be, OK, I'm going to set the x and the y as follows. I'm going to say y_hat is theta1 times x. And then we'll return the mse_loss with x and y_obs. Why is this any different? Why is this useful? Well, now if I want to run this sort of guess-and-check experiment, instead of having to enter in x and y_obs every time, I only need this value, right? So, if I plug in 0.2, I get 2.667. If I plug in 0.1, I get 2.077. If I put in 0.1437, I get 1.178. OK. So, this right here is a single-argument version—hence the name—of the mean squared error. OK.

So, why is that useful? Well, we can of course minimize this as we did in an earlier lecture by just brute-forcing, or actually as we did earlier today even, with our arbitrary function. So, mirroring what we did before, if we plot that mean squared error as a function of θ_1, we see that it goes down and then back up, with a minimum of around 0.1437. And so, this specific curve we're getting, is a function of all the blue dots. So, if we wiggle these blue dots around—like drag some up and down and over and so forth—we will get a

slightly different curve. But it will always be a nice bowl-shaped curve, like so.

OK, so it'll look like this. And so, we see that the best θ_1 is somewhere around here. So, in other words, rather than trying to optimize our earlier function, which was very wiggly. You remember this thing? OK? This one was very wiggly. Now we're minimizing a much simpler bowl-shaped curve, but it's the same basic idea. OK. So, this is the curve we're going to minimize. And so, one way is, we could do exactly as we did in this module, where we just try out a bunch of guesses and see which one's best. We can even visualize that procedure. And we try out a bunch of values. So, that's one approach we could do. And in fact, when we talked about linear regression earlier, we outlined this approach.

Now, of course there are better ways. One thing we could do is, we could just optimize. We could use the scipy.optimize function and say, here is a mean squared error function that takes a single argument. Let's start from 0 and it finds a value, which is 0.1437. So, in other words, the thing that scikit-learn does is basically this: It sets up a single argument mean squared error and then passes it to a minimization library that does the actual hard work of finding this value. Now, let's do it ourselves using the gradient descent function that we created earlier.

So, recall, here is our gradient descent algorithm that we wrote together. And the only difference here is I'm using the variable named guess instead of current guess. That's not important, but just how I wrote it here. And so, I'm going to use this function to find the optimal tip. So, in other words, rather than relying on somebody else's code to come up with this value,

we're hand rolling it totally from scratch to get the answer that we desire. Now to do this, we cannot just provide our mean squared error function to our gradient descent function. It requires the derivative. And so, what we need to do is write the derivative of this mse_loss function. That's the thing we're trying to find the minimum of. So here, correct a variable name here. I think theta_1 is a better name. The derivative of this function is going to be as follows.

So, instead of it being the average of the difference between our predictions and the observation squared. Well, if I take the derivative of this, we bring down the 2 and we end up with 2 times y_hat minus y_obs, all times x. Where did this x come from? Well, that's from the chain rule. So, we also need to take the derivative of y_hat, which is going to be just x. So, the derivative of theta_1 times x, with respect to theta_1 is just x. OK? So, keep in mind, by the way, that our derivative here is not with respect to x, but is with respect to θ_1. And that's actually a really important point for me to emphasize. What we're optimizing is the parameters, not the data. And so, we're taking the derivative with respect to theta_1, which yields this expression. So, once we have this expression, mse_loss_derivative, this is basically just like we had before, with our derivative of the arbitrary function.

So let's try out some different values. So, if theta1 is 0.1, x is the tips...sorry, the total bills, and then y_obs are the tips. In that case, we can ask for the derivative with respect to theta1 if theta1 is 0.1. If I pick a different theta1, we get different derivatives just as before. Like this is just the same thing we saw with the derivative of our arbitrary x^4 function. And so, when we're running the gradient descent procedure, what this tells us is that when

theta1 is 0.1, the derivative is negative. Right. So, that's something we'll use in order to generate our next guess for theta1, just like before. And so here, what I'm going to do is—just like before when we did our mean squared error of theta1 function—we can change our function to be just a function of a single argument.

So, this mean squared error loss derivative function here, rather than having these kind of silly arguments that don't do anything, I'm just going to create a single-argument version. OK. And again, the reason I do that is because it makes it more syntactically convenient. So now, instead of having to plug in all three of these values, I can plug in just 0.1, OK. And so, here I can plug in various values. And by the way, if I plug in a value near the minimum, you'll see the derivative is very small.

OK, so now it's time to work our gradient descent magic using our function that we wrote ourselves. So, all I'm going to do is I'm going to call gradient_descent—which I wrote—on mse_loss_derivative_single_arg, which I wrote. I'm going to pick a learning rate, and a starting value, and a number of points. Sorry, this is the learning rate here. And so, we're going to start with giving them a 5% tip, have a relatively small learning rate, and we'll do a 100 guesses. And so, what we see is the original tip that the algorithm decides is 5%. Because that's what we told it. And it says, well that's too low. Like the derivative tells me I need a bigger θ. So, then it says 5.88. OK. Then it says 6.6. And it keeps trying different tips, over and over, until eventually it converges to some value.

And so, what we see here is that we have written our entire end-to-end implementation of gradient descent and indeed linear regression with

nobody else's code other than, I guess we use someone else's mean function. But we did all the hard stuff ourselves. Now, why didn't we do this at the very beginning when we talked about linear regression? One of the great things about computer science and about AI and machine learning and all of this, is abstraction barriers. We don't necessarily need to care about every single little detail all the time.

Before I told you about loss functions, the only thing I hadn't told you is how gradient descent works. So, now that we have some context and we understand why all of this machine learning stuff is actually useful, I'm peeling back the veil. But that doesn't mean that you want to write your own gradient descents in the future. The goal of this module is to give you an understanding of how these things work, so that if something is behaving strangely or isn't working how you expect, you have some insight into how the algorithms work. But I should warn you, it's usually a bad idea to write your own optimization libraries because there's a lot of subtle tricks that you need to use in order to get really, truly good performance.

Now, I should also note that we can actually write out what just happened, also as a recurrence relation. And just to get a sense of what that looks like, the way that our expression works in this machine learning context, is the next θ is the old θ minus the learning rate times the derivative with respect to θ_1 of the loss. Where the loss depends on not just the guess, but also a hard-coded, or just fixed dataset and a set of observations. And so, that right there parallels exactly what we saw earlier when we derived gradient descent. But here it is in a machine learning context.

So, now hopefully you have some intuition for how gradient descent actually allows libraries like scikit-learn to do their job. But I should remind you, we're never really going to write these gradient descents on our own. But it's good to know that the gradients exist. And in more advanced contexts, often you'll actually end up writing the derivative functions out yourselves manually. And you'll see that if you ever go into more advanced fields like, say, neural network design.

## Video 6: Convexity

As we just saw, gradient descent was able to give us the same optimum parameter as scikit-learn. In other words, it is able to find the global minimum of the mean squared error function. This raises the interesting question: Does gradient descent always give the optimal linear regression parameter? As we saw earlier with our arbitrary function, gradient descent can get stuck in a local minimum. However, if a function has a special property called convexity, then gradient descent is guaranteed to find the global minimum.

Normally, we'll say that a function, f, is convex if—and only if—it obeys this inequality. t times f(a) + (1 minus t), times f(b) is greater than or equal to f(t times a) + (1 minus t) times b for all a and b in the domain of f and t in the range between 0 and 1. Now, that inequality probably seems pretty mysterious at first. And on the homework, you'll have a chance to build a deeper mathematical intuition for what this formula means.

I can also give a simple English translation of this formula. If I draw a line between two points on the curve, all values on the curve must be on or below the line. Now as it so happens, the mean squared error loss function

in one dimension is convex. So, for example, if we draw this line segment between these points on the mean squared error curve, we see that all the values on the curve are below the line segment we drew. And indeed, if we draw any such line segment between two points on the mean squared error function, all the values on the curve between those two points will be below that line segment.

Though we will not do so here, one can show that for any data set, the mean squared error is always convex. Why do we care about convexity? One reason is that for any convex function, f, any local minimum is also a global minimum. That means that the loss function is convex, then gradient descent will always find the globally optimum parameter in one dimension, or a set of parameters in higher dimensions. Now by contrast, consider our arbitrary function from before with two local minima.

Here, I've drawn a line segment between two points on that curve such that not every point on the curve is below the line segment. By the definition of convexity, this arbitrary function is therefore not convex. Thus, gradient descent may get stuck in that local minimum to the left. A second reason we care about convexity is that optimization is generally much easier for convex functions.

So, as an example, consider the two, two-dimensional functions shown. The left function is non-convex and the right one is convex. For the left function, if you're far from the minimum in that large, relatively flat region, the slope is too shallow to efficiently follow. Only very near the minimum, is the slope steep enough to follow downwards toward that minimum. By contrast, the convex function, it has enough steepness anywhere so that you can work

your way down towards the bottom no matter where you start. And on the homework, you'll generate and visualize very similar curves.

## Video 7: Optimizing 2D Linear Regression Using Non-Gradient Descent Techniques

Let's now generalize our model to have not just one parameter, but two. So, specifically what I'm going to do is for each prediction, I'm going to assume that the tip is going to be the original bill times the tip percentage, as before. But now I'm going to add this θ_0 offset. So, in order to do this, I'm going to again reload the tips dataset from scratch, just to show it on the screen. And then I'm going to fit my model. I'm going to say that we're going to include now an intercept, that's going to be θ_0. And so, when I fit this model and I ask for the coefficient and the intercept, I get back a tip percentage of 10.5% and an intercept of $0.92. That's the way this model shakes out.

So, what I want to do in this video and then the next one as well, is I'm going to compute these two values myself from scratch without help from any other library. That's my ultimate goal. Now, to make my life easier, for reasons that will become clear, I am going to reframe this question slightly. So, what I'm going to do is, I'm going to create a new copy of the tips dataset. And I'm going to create a bias column of all ones. OK. So why, what's that all about? Well, now because I've done that, I can tell my linear regression model that I want to not include an intercept. And instead, this bias column is going to effectively give me an intercept.

So, in other words, if I fit this model and then I ask for the coefficients, the intercept is θ_0 and the slope is θ_1. OK. So, what's going on there exactly? Why does that work? Well, this is a common trick that you'll see in all kinds of models out there. The reason this happens is that, if we think about our predictions, they are θ_0 plus θ_1 times the bill. And so, this piece of code right here gives me predictions.

So, for example, if θ_0 is 1.5 and θ_1 is 0.05, then my predictions are just θ_0 times X, the first column of X, plus 0.05 times the first, or the next column of X. Right. So, if I do that, I get a bunch of predictions. Now of note, I don't actually need this because this column is all ones. And so, if I delete X.iloc[:, 0] I get the same predictions. OK. So, this is just an alternate framing for how to do the exact same problem. But when it comes to the gradient descent step, it'll be nicer to have everything in one framework where we're not thinking about the intercept as some totally separate type of parameter. This is just a little cleaner to have them in the same coefficient array. OK.

Now, for those of you who want to know, I'll mention there's a nicer way of writing this code. So, instead of saying 1.5 times the 0th column plus 0.05 times the 1th column and so forth. I can just say, create the matrix multiplication of X and this array and you'll get the same thing. Though, I don't really want to go into too much detail there. But I'm just mentioning it because it is nicer practice. But for our purposes today, it's actually easier to read the code if I exactly mimic the form of the equation as we wrote above.

So, this is the form I'm going to use, but technically you can write it this way as well. OK. So, hopefully now you understand that I can create a linear regression model where we have a bias column and each prediction is simply θ_0 times that bias column + θ_1 times the actual data—the X's that I care about. Now, I want to create a mean squared error function, which is what I'm going to be optimizing against. So, the mse_loss, given a choice of theta, where there's two different components to theta. And the data X and the y_obs is going to be as follows.

First we compute our predictions, and then we return the difference between our predictions and the truth, all squared. And then computing the mean. And as before, we could use the scikit-learn metrics that computes the mean squared error. But here I'm just being as explicit as possible. And so, once I've written this function, I can then plug in values for θ_0 and θ_1 and see what happens. So, if I pick a fixed tip offset of a $1.50 and a 5% tip, this is my mean squared error loss. If I make this $2.50, I get a slightly different loss and so forth. If I change this to a 0% tip, I get a larger loss.

And so, this right here, already, you can anticipate that we could use this to find the optimal θ_0 and optimal θ_1. We just plug in a bunch of values and see what happens. We use gradients, something like that. And so now, let's repeat the same idea as before, but now on this two-dimensional function—all these different approaches. OK. So, the first approach is brute forcing. In this approach, we're just going to try a bunch of different theta values. And this right here is the same exact mse_loss from above. I'm just copying it here for clarity. So, this is the function that gives me the mse_loss. But as before, it's convenient to create a version of this function that only has one argument.

So, how do I do that? Well, I just set aside X and y in a new function, that's single argument, and then return. This version actually requires these explicitly. OK, so we're going to use that older mean mse_loss function that has an X and a y_obs and return the result of that function. So, this is just a wrapper. Right. A convenience function that makes life easier. So now, if I want to try out different parameters, I can only change...I can only provide this one argument and use this to explore my space. I don't have to have the X and the y_obs, because it's the first thing I've done.

So, this is our launching point where now we can finally optimize this thing. And what we'll find is that the optimal choice will be something around 92% and 0.1. OK, so how do we optimize and find that these are the right values? What we're just doing the same thing as before. So, we could create a plot of this loss function as a function of its two parameters. And in this case it's going to be a three-dimensional loss surface. So, I don't want you to focus too much on all of this code, which is something that I put together to generate this loss surface. And you'll see this right here, is the error of our model as a function of our two parameters. So, one is our fixed offset tip, that's theta0, and the other is our percentage. Right. And so, somewhere right here in the bottom of this bowl shape, we see that x of 0.8888 and y 0.111, is the optimal point.

Why those numbers? That's not 0.92 and 0.105 or whatever. That's because I only looked at a 20 by 20 grid of points. Actually, it's a 10 by 10 grid. So, I just didn't happen to hit the, the true minimum, right, just as before. But it's pretty close. So, we just try out 100 different values and this was the best one. And so, what our gradient descent procedure and our optimization

libraries are doing, is basically rolling their way down this two-dimensional function now towards this minimum.

OK, let's use one of those optimization libraries then. Right. I used scikit-learn. But what I can do is, actually, I can give it this mse_loss_single_arg function—the one up here that we were playing with manually. So, if I call that function starting from a 0% tip with a $0 offset, you'll find that it actually optimizes and finds $0.92 fixed offset and 10.5% tip. In other words, it used gradient descent to find the minimum of this function right here. And so, that's the off-the-shelf library version. And next, we'll see how we can do this ourself using our own gradient descent code.

## Video 8: 2D Gradient Descent

Let's now turn our attention to convex optimization on multivariable functions. We'll use this procedure to minimize functions like this 2D loss function for the tips dataset, where the two inputs are the parameters theta0 and theta1. As we'll see, multi-dimensional gradient descent has some important differences from the one-dimensional case. Just like with one-dimensional functions, we can optimize a 2D function by following the slope. However, unlike a 1D function, the slope of a two-dimensional function is described by a two-dimensional vector. In other words, the best way down has two components, each corresponding to the slope with respect to the two input variables to the function.

Consider this wireframe picture here of a two-dimensional function. If the star represents our starting point, the arrow shown gives the best way down. If we were doing a gradient descent, the size of the jump we make will be based also on the learning rate alpha. We will use that alpha, along

with the two values representing this 2D arrow, to reach our next guess for the optimal parameters. After making our first jump, we compute the best way down again, yielding this new arrow shown. And we again follow the arrow down. We again follow the arrow down, now having made three leaps. After the third leap, we see that the best way down is now starting to curve off from our original direction. Thus, our next leap will take us in this new direction. After our next jump, the best way down is curved off even further. We'll repeat this process over and over until the function we're optimizing is no longer decreasing significantly with each step.

The formal term for the best way down, is the gradient of the function. The gradient, which you may have studied in a previous class on multivariable calculus, is the generalization of the idea of a derivative of a one-dimensional function. Each component of the gradient represents the partial derivative of the function, with respect to one of the input variables to the function.

So, for example, consider this 2D function f, which has input variables θ_0 and θ_1. This function equals 8 times θ_0^2 + 3 times θ_0 times θ_1. Note that this is not a linear regression loss function, as it has a nonlinear dependence on the thetas. This is just some arbitrary function of θ_0 and θ_1. Now since this function is two dimensional, its gradient has two components given by the formula shown. The symbol used for the gradient is the nabla, an upside-down triangle. And this upside-down triangle symbol is also often pronounced as del. For example, here we see that the gradient del sub θ of f is equal to the partial derivative of f, with respect to θ_0, times the unit vector i, plus the partial derivative of f with respect to θ_1 times the

unit vector j. Here, when I say the unit vector i, I mean the unit vector in the θ_0 direction. And the unit vector j is the unit vector in the θ_1 direction.

Given this definition, we can compute the gradient of our function, f. First, we compute the partial derivative of f with respect to θ_0. This is 16 times θ_0 + 3 times θ_1. Then we compute the partial derivative of f with respect to θ_1, and that's just 3 times θ_0. So, given these two partial derivatives, we can then say that del sub θ of f, is the unit vector i times 16 + θ_0 + 3 times θ_1 + the unit vector j times 3 times θ_0.

Now for notational convenience, gradients are often written in column vector notation. For example, we can also write the gradient as, del sub θ of f(θ) is equal to big square braces with 16 times θ_0 + 3 times θ_1 on top, and three times θ_0 on the bottom. This is just the same thing in different notation. We can expand this definition of the gradient to p dimensions. So here, the top value in our column vector is the derivative of f with respect to θ_0. The next value is the derivative of f with respect to θ_1, and so forth. Until we get to the last row, where we have the derivative of f with respect to θ_p.

Let's now discuss how we can use the gradient of a function to find its optimizing parameters. In the context of model optimization, we can think of the top value in the column vector as follows. If I slightly increase θ_0, what happens to the function, or what happens to the loss? The next row tells us if I slightly increase θ_1, what happens to the loss and so forth. So, if we want to reduce the loss, we should increase all of the θs that have a negative partial derivative. And we should decrease all of the θs that have a positive partial derivative.

So, this finally brings us to our gradient descent algorithm. To get the best possible improvement, relative to our current guess, we should increase our θs proportionally to the magnitudes of the partial derivatives. Since that is the multidimensional vector that represents the steepest direction of the slope at the current point on the loss surface. So, our gradient descent algorithm will nudge our θ vector in the direction of the negative of the gradient. And just as before, we will...we'll repeat that process until eventually θ converges.

Now, written in mathematical notation, we can say that the next guess, θ_(t + 1) equals θ^(t) minus α times del sub θ of the loss function. And here, the gradient is given by del, α is the learning rate, θ^t represents the current guess for optimal parameters, and θ^(t +1) represents the next guess for the optimal parameters. Note that the gradient of the loss function depends on our current guess for θ, the input features X, and the true value we're trying to predict, y. Also note that when we optimize a function using this gradient descent procedure, we have to have some sort of starting guess. This could be all zeros, small random numbers, or some arbitrary starting guess provided as an input to this gradient descent procedure.

## Video 9: 2D Gradient Descent for Linear Regression (Part 1)

So, we've seen how to define the gradient of a function. But to get a better understanding of just what that definition means, and also to see why it's useful, let's now see how we can write our own gradient descent code from scratch and use it to optimize a two-dimensional linear regression model. So, the first step is, we will need to compute the gradient of the loss for linear regression. So, here we have a two-dimensional linear regression

model with no intercept term. And the output of that model is θ_0 times x_0 + θ_1 times x_1. And we know that of course, the loss for such a model—if we're using the squared error—is y_i minus our prediction squared. And that's a loss for just a single data point.

So, if we substitute in our model prediction, we have the loss for a model equal to y_i minus θ_0 times x_0 minus θ_1 times x_1 all squared. Here, this equation, it gives us the loss only for data point number i. Now at this point, I want you to pause the video—and I strongly encourage you to actually do this—and give me the gradient of this loss function for data point number i. Keeping in mind that the gradient is going to have two components.

Okay. So, I'm going to spoil the answer for you now, so hopefully you tried it. So first, we need to compute the first component of our gradient. So, that's the partial derivative of the loss function with respect to θ_0. So, what's that? Well, that's just 2 times y_i minus θ_0 times x_0 minus θ_1 minus x_1, all times minus x_0. Now, that minus x_0 term, it appears because of the chain rule in calculus. So, for the second component of the gradient, the partial derivative of the loss function with respect to θ_1 is, 2 times y_1 minus θ_0 times x_0 minus θ_1 times x_1, all times minus x_1.

In other words, this component of the gradient is exactly the same as before, only now with minus x_1 instead of minus x_0. We can also write that as a column vector, giving us del sub θ of f is equal to the column vector shown, where the top row is the partial derivative with respect to θ_0 and the bottom row is the partial derivative with respect to θ_1. So, now that we've derived this expression, Let's put this into code.

## Video 10: 2D Gradient Descent for Linear Regression (Part 2)

So, here is the equation we derived for the gradient with respect to θ_0 and θ_1 of our loss function. So here, this top component is the derivative with respect to theta 0, and the bottom is the derivative with respect to theta 1. So, let's write that out in code, and we will be using this with our gradient descent function.

So, first I'm going to set aside the x0 is going to be the 0th column of our array. Remember actually, by the way, that that's just all ones. And this is going to be our bill. So, that's x0 and x1. And then I'm going to write out these two expressions. So, in particular, I guess I'll say the derivative with respect to θ_0 is going to be minus 2 times the y_obs value minus θ_0 times x_0 minus θ_1 times x_1. And then all of that is going to be times x_0. OK. So, that's our first component. And then the second component of our gradient is going to be θ_1, same thing except here I'm going to have x1. And then once I've computed those two components, I'm just going to return a NumPy array of those two components, θ_0 and θ_1. OK. So, that's literally just what we derived now written out in code.

Let's test it out. So, if I didn't mess anything up here, when I call this function, I should get back some values. A gradient, that's just going to be two values. Now there's one thing that I forgot to do, which is that I also need to keep in mind that we're supposed to be computing the averages. So in fact, I should be saying np.mean here. And this will give me back the values that we want. OK. So, now that we do this, we get back a gradient here. So, the average that we get if we take into account all the data points, is as follows. Basically, what it tells us is that if our original guess is that the

fixed offset is $0 and the tip percentage is 0% that, well, we need to adjust both of these upwards because the gradient is negative in both directions. OK.

Now of course, it's convenient to write a single argument version of the gradient descent function as follows. So, this is the exact same trick as before, Where we're going to hardcode in the X and y values and then just return the output as a function of a single argument θ. Now keep in mind this single argument is both of our θs, so it's an array of two numbers. So, now if I try out a value like [0, 0] I get back minus 5.99996 and minus 135.22.

OK, so this is the derivative, or rather the gradient, of our two-dimensional loss function. So, I put...plug in this value 0, 0. That's telling me the two-dimensional slope of that original loss surface. In other words, if we scroll back up, that's telling me the slope of the value once we're at 0, 0; though that's not actually shown on the page. And so, if I plug in something close to the correct answer, that is 0.9 and 0.1, you should see a gradient, which is pretty small.

So, let's test that out. So, if I do 0.9 and 0.1, we should get a relatively small gradient and indeed we do, right? We can see that these two values are much smaller than for some arbitrary guess. Now comes the gradient descent part. OK, so the cool thing about gradient descent, is I don't have to write different code. The gradient descent function that you and I wrote earlier, that same five lines way up higher. Actually, let's go look at it. So, if you look at def gradient_descent. This function right here, we're going to use the same one. It doesn't matter that before we were doing one-

dimensional and now we're doing two-dimensional. It's irrelevant because of the way that NumPy works in Python. Right. So, now the difference is that the df function we're passing, is now a two-dimensional function. But this code right here still works just fine.

So, if we go down to the code we were just writing, we can just run gradient_descent. Now I'm actually not going to run it right now because it takes a while. So, if I give it the number of iterations I need, and the number...the learning rate that actually works. It would take about, I don't know, a minute, maybe 30 seconds. And that's just a little too slow the way that the code is written right now. However, after running it, and you're welcome to do this on your own time, if I look at the trajectory of the procedure of gradient descent, we see that at the beginning, it moves pretty quickly. It starts with a $0.00 offset and a 0% slope. And it increases that offset and the slope jumps around. OK, so it's evolving over time. And then at the very bottom, once it's done, you can see that it's getting pretty close to converging to the correct answer. It's still not quite there, but it's much closer. $0.888 offset and 10% tip. Now if you play around with this code above and you try different learning rates and different starting points, you'll see that it's pretty finicky. And so, for that reason, if you want to write a true industrial strength gradient descent, you got to be a little more careful than the way we have. But actually, our code ends up working as you see here, and converging to the correct value.

So, what we've just done now, what's kind of cool about this, is that we didn't use anybody else's libraries. And we managed to find the answer to; What is the best two-dimensional model, with both θ_0 and θ_1, that gives us the correct answer? Now, what we haven't done here, is we haven't done

the visualizing of our trajectory across this loss surface. But you'll have a chance to try that out on this week's homework.

## Video 11: Stochastic Gradient Descent

Real gradient descent implementations these days use another trick. They result in something known as stochastic gradient descent. Let me give you some intuition for why this approach is used, as well as a rudimentary implementation of this algorithm. So, recall that our gradients are going to be a function of the entire dataset. So, in other words, if we want to know the mean squared error across our entire dataset, we have to compute the difference between our prediction and the actual tip for the entire dataset.

Now here that's not such a big deal because there's only 244 rows. But in real datasets where maybe we're doing image processing or something like that, or image classification. In those cases, we may have millions or even billions of data points and it's just not practical to be computing the gradient on all of those data points. OK so the idea is, I can't run the code I had from before, where we just look at every single data point. So, we want to do something a little different.

So, here's a somewhat curious idea. We can instead compute the gradient only taking into account some of the data, right? So, instead of using the gradient across everything, we'll just pretend some of the data doesn't exist. OK. So, for example, the gradient of our mean squared error with respect to our two parameters across the entire dataset is going to be this value we computed, but that value is kind of expensive. So instead, I'm going to create a function here called mse_gradient_batch_only, where you're going to give me a set of indices. That is the batch of data I care about. And that's

going to give me the gradient if I only take into consideration a subset of the data.

So, it's very simple code—if you understood the previous code—where instead of x0 being all of the entries in column 0 and x1 being all the entries in column 1, I'm only going to take the rows you'd give me. OK. So, then when I actually compute the errors, I'm again only going to consider the observations that I specify. And so, now if I write this function and I say, give me the gradient for only the points between 0 and the length of the array. Well, in this case, if we run this, we should get back that same value as above. Because this is saying, take into account everything.

But here's the cool part. Now, I can say: What if I only want to know the loss on point number five? So, what that's going to do is it's going to tell me the loss only taking into consideration the fifth column here. Alright, so that right there is much faster to compute compared to the entire dataset. It'll take about 1/244th of the time. OK. Now I can also take a subset of data, like a batch of data. So, I can say give me everything between row 0 and 4, let's say. And that'll give me this value here. So, that's the average gradient that I get back across only those data points. OK. Now that might seem weird, right? The idea that this gradient would be useful considering only some of the data points. And so that's where stochastic gradient descent will come in. OK.

Now, one thing I want to do before I introduce the...the algorithm is I want to create an only_two_arg version of mse_gradient_batch_only. So, rather than having to provide X and y_obs, I'm going to do the same old trick as before, where I'm going to hardcode these. And so now when I want to know the

gradient, I'm only going to provide the two thetas and the batch indices I care about. So, only index 5 or indices 5, 6, 7, 8. I also want number 15 and number 32, right. So, that lets me compute the average, the gradient across only the values I specify. OK.

So, given all of that, I'm going to show you an interesting trick. So, the next piece of code I'm going to write is going to hinge on the behavior of this NumPy split function. You're given an array of values like, say 1 through 9, and you give it the value 3. And it's going to split them into three equal parts. So, if I give it, for example, the split of a random permutation, of np.arrange, let's say 10, And we'll split...actually do 12. And we'll split that into three parts. I get. And three arrays, which are: [11, 8, 10, 1], [6, 0, 5, 7] and [9, 2, 4, 3]. OK.

So, right now you should probably be a little confused. You should understand that what this function does, is it gives me the gradient only taking into consideration a subset of the points. And you should also understand that this line of code splits an array of numbers between 0 and 11, into three, different sub-arrays. OK. So, this brings me to stochastic gradient descent. What this algorithm does, and I'll show it in code first and then later we'll, I'll...in the next video, I'll show it to you in symbols and in an equation form. What it does is the following. So, it's the same idea as before, except this time I'm also going to keep track of my losses as I go. And so, while the number of guesses is less than n, what I'm going to do is create a bunch of indices. Different arrays of indices, like these right here, and set them aside.

So, when I say number of batches here, what this...and is, right here, is the number of data points. This is going to create an array that's between the numbers, say 0 and 12, right? So, this is actually, maybe I'll make this extra clear by putting this on a line above. So, let's say you have 12 data points. This will create a random permutation of the data points. And then this line will split it into separate arrays. OK. And so, then what we'll do is for each of these arrays, it's going to do the following. It's going to compute the gradient only with respect to the single batch here, this little mini batch of four data points. And then it's going to apply that result to the guess. That is, it's going to compute a next guess based on only, in this case, four of the data points. And then it's going to append that guess and then also add the loss to this record of all the losses. Then it is going to repeat that process for data points 6, 0, 5, and 7. OK? And then it's going to repeat it one more time for 9, 2, 4, and 3.

So, the basic idea is that rather than doing one gradient descent step across, say, all 12 of these data points, it's actually breaking it into three rounds. The first round considers only four of the data points. The next round considers only another four of the data points. And then, finally we consider only the last four of the data points if we had 12. OK. Now, in our case we actually have 244 data points. And I'll pick a batch size of say, or I'll say that I want four batches. And so, it's going to be four batches of data, which are each size 61.

OK, so that's basically how our gradient descent procedure is going to evolve. OK, so if I run this code with learning rate 0.001 and I say, do 10,000 steps. I start from [0, 0]. I run it on my entire tips dataset. And I say, give me four batches. It's going to do 10,000 steps, where each step is going to be a

pass over each of these four subsets. And every time it goes through, it's going to generate a different permutation of the data. So in effect, we're going to be having an approximation of the gradient, rather than using the true gradient. And the argument here is that each of these gradients is more efficient to compute. OK.

So, what happens if I actually run this code? Well, just as before, it's going to be a little slow. But the result is as follows. So, here I have the results of the experiment. And the reason that I track the losses, is so we can see the pattern of the losses as we go. And here I have the theta0 and theta1 values over time. Let me go up here real quick and show you the top of this results frame. So, at the very beginning, the first theta0 and theta1 I have, are 0 and 0 and the loss is relatively large. And then after that, we make some jumps. And the loss is better. And over time, as we run this algorithm further and further and further, we end up with different theta0 and theta1 values.

Now, the pattern here is a little different than before because the…each step of the algorithm is only using a subset of the data. And so, it seems plausible that maybe that subset is missing some outliers somewhere. And so the, the gradient descent algorithm may make steps that are not really as globally optimal as the gradient descent where we're using the entire gradient.

Now, I should say this is one of the trickiest concepts that we will cover. And so, I don't really expect you to fully understand it in this lecture video. But I hope that between this video, the next video, and most importantly the homework, that you could be able to digest this interesting tweak on gradient descent that we call stochastic gradient descent.

## Video 12: Gradient Descent vs. Stochastic Gradient Descent

Here we see a formula for gradient descent in mathematical notation. We see that computing the gradient requires iterating over the entire dataset. So, for our example, this isn't such a big deal since the tip dataset is so small. However, for larger real-world datasets of millions or even maybe billions of data points, computing the gradient can be extremely costly. In the approach that we just saw, we instead compute the gradient for a batch of data where the batch is smaller than the entire dataset. In the case where the batch size is one, we have stochastic gradient descent, where the optimizing algorithm is only considering a single data point at a time. It should surprise and perhaps even shock you that this works.

Imagine that we're trying to train an algorithm to recognize pictures of dogs. Stochastic gradient descent adjusts what might be an absolutely, massive number of model parameters, based on only a single training image at a time. The analogy might be something like trying to win a game of chess, while only looking at a few squares of the board at once. And yet, stochastic gradient descent works. How that happens is beyond the scope of the class. But the rough idea is: If we iterate over the entire dataset multiple times, well, then on average we'll end up in roughly the same place as if we'd been computing the true gradient based on the entire dataset.

In effect, our batch size is a parameter that gives us the ability to tradeoff the quality of our gradient approximation against the runtime to compute that gradient approximation. If the batch size is one, the quality...the quality is minimum, but the calculation is very fast. And if the batch size is the entire dataset, well, then the quality of the gradient is extremely good, but

the calculation may be very slow. There have been many rules of thumb developed. Pick a batch size, for example, for reasons that are far beyond the scope of our course, 32 is a common choice of batch size.

So, let's see the idea from another perspective. Here we see a visual picture of gradient descent in two dimensions, where our batch size is the size of the dataset. Here, the darker the color, the lower the loss. We see that if we start at the origin, gradient descent follows the steepest path towards the minimum and it's the true steepest path. Now, we see a visual picture of gradient descent, where our batch size is small. So, same figure because it's the same loss surface, but we don't compute the exact loss surface as we go. So, in other words, this time—since we're only considering a small subset of the data—we're only getting a crude approximation of the gradient.

So, for example, you'll notice this first jump is not in the direction of the steepest drop towards the minimum, but it does give us some progress relative to where we started. We repeat this process again and again and again, moving in a somewhat random fashion across the true loss surface. After all, at each jump, we're only moving based on a crude approximation of this true loss surface. Eventually for convex functions, like the mean squared error, we'll end up converging somewhere very close to the true minimum.

Now in practice, both techniques are used. For example, for all of the scikit-learn models we've talked about in our class, graded...gradient descent on the entire dataset is what the library uses. However, in environments with large amounts of data, it is much more common to use mini-batch gradient

descent, because the cost of computing the gradient on the entire dataset is too high, resulting in very slow algorithm training times. Now zooming out a bit. In practice, you will almost never write your own gradient descent algorithms. And for the most part, you'll be using function optimization libraries without having any need to know how they really work.

Indeed, until this module, we've just treated our function optimization libraries like magic. In the same way that you can drive a car without understanding how an engine works, you can train and even use machine learning models without ever learning about gradient descent. Since, in some ways, that optimization procedure is a low-level detail that is handled for you by somebody else's code. Now however, in my opinion, having at least some knowledge of how gradient descent works is important for serious machine learning practitioners. Even if you're not interacting with gradient descent directly.

## Video 13: Implicit Regularization and Stochastic Gradient Descent

Recall that our classical picture of machine learning models is that as model complexity goes up, the training error will tend to decrease more and more. By contrast, the test and validation error, they go down for a while and then they go back up as the model begins to overfit. Now, here's the same picture again from a paper called *Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-Off*. This is a paper from 2019 by Belkin et al.

So, here they use slightly different terminology. The error is called risk and the complexity is called the capacity. These two terms have a slightly different technical meaning, but that's beyond the scope of our course. Now, in the late 2010s, it was observed that some models exhibited a surprising relationship between the test error and the model complexity. This figure, also taken from that same paper, *Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-off*. It shows a common pattern that was observed when training large neural networks.

So, right around the level of complexity that results in zero training error, the test error spikes. But, if we push the model to even higher levels of complexity, the test error starts to decrease again. And in many real-world cases, that test error actually ends up reaching an even lower value than the sweet spot that we discussed in earlier lectures. In other words, in the world of practical machine learning, an entirely new regime was discovered for large models, which this paper calls the 'Modern' interpolating regime. They use the word interpolating to represent situations where the training set has zero error. Now, other papers from this same time period showed this phenomenon in many other machine learning contexts, even very simple ones. And that includes even linear regression.

These figures, taken from a series of tweets by Dr. Daniela Witten in August 2020, portrayed the situation quite nicely. So, first here, we see a degree for linear regression model fit to 20 data points. Note that here Dr. Witten is using something called spline regression, which is similar to but not quite the same thing as polynomial regression. So, here the original data points are in gray. The blue line is the fourth-order model fit. And the black line is the real process which generated the data, which also had some noise

added in that gave you the gray data points. Now, when we move to a degree-six model, as you expect, we would do even better. But if we now move to a degree-20 model, well, we see that we have zero training error. But our model is grossly overfit. And this is very similar to what we observed in our class. The model is interpolating the observed training data points perfectly. But for points adjacent to the training data, the model is failing to capture the true data generation process. It's a mess. We've clearly overshot the sweet spot. Now, here's where it gets kind of weird.

What if we have even more than 20 parameters? So, for example, imagine fitting a degree-36 polynomial, to 20 data points. If you think about it, we basically have 36 linear equations and 20 unknowns. And thus, there's an infinite number of possible solutions that all yield zero training error. This figure, also by Dr. Witten, shows the side-by-side results for a 20-parameter and a 36-parameter model. The 36-degree model is slightly better. By moving deeper into the modern interpolating regime, we've now observed double descent, where the test error actually drops again. If we plot Dr. Witten's training and test error versus complexity for this example, we get the figure shown.

Note that the test error in this case spikes back up for even more degrees of freedom. In other contexts, like in neural networks, this spike back up often doesn't occur. Also note that the test error after the second descent does not reach a new lower value. However, in some contexts, like those shown in the cartoon earlier from Belkin's paper, the test error is actually lower after the double descent and the interpolating regime. So, what—you may ask—does this have to do with stochastic gradient descent? Which is what we're talking about today. Well, it turns out that understanding

stochastic gradient descent also gives some insight into why double descent occurs.

So, let's rewind a bit. As Dr. Witten notes, there are an infinite number of models, which are over-parameterized and have zero training error. As we showed earlier in our class. If we have n data points and a model with n parameters, then there exists exactly one set of parameters that fits the data perfectly. But, if we increase the number of parameters even higher, we end up with a situation where the model is over-parameterized and there are an infinite number of valid choices of parameter that give you that zero training error. So, how do we know which model we'll end up with from amongst this infinite sea of possibilities? Well, that's where knowledge of stochastic gradient descent becomes useful.

Stochastic gradient descent is a specific procedure. So, we can experimentally explore well...where it will land among all of the infinite possible models with zero training error. Now, for reasons beyond the scope of our class, it turns out that stochastic gradient descent yields a solution, which is implicitly regularized. In other words, even without an explicit regularization penalty, over-parameterized models that are trained with stochastic gradient descent act like they are regularized. And as the number of parameters in a model grows, the impact of this effect increases. The net result is what we saw in Dr. Witten's post, where extremely high-order models are actually less wiggly. Or, more plainly stated, stochastic gradient descent results in a model, which is implicitly regularized, yielding less wiggly behavior.

Counterintuitively, because the 36-degree model has a greater implicit regularization strength, it is actually less free than a 20-degree model.

For a deeper treatment of this tricky idea, see the 2021 paper *Implicit Gradient Regularization* by David Barrett. I should note that at the time of this video, the double descent phenomenon—where the test error drops again—and other related topics are still a topic of very active research. And thus, a full treatment is beyond the scope of our class. I'm presenting this to you today to give you a flavor of the ideas. But I do not expect you to deeply understanding these cutting edge notions or the underlying theory that I do not myself fully understand yet.

## Video 14: The Bias–Variance Tradeoff

Before we wrap up this module, let's briefly discuss an important concept known as the bias–variance tradeoff. We can think of the error of a model as stemming from two sources: The bias and the variance. The bias represents the fundamental inability of a model to fit the data, no matter what parameters are provided.

For example, consider the Miles per gallon dataset shown. Suppose we fit a model, which only has an intercept term. In other words, it believes that all vehicles have the same fuel efficiency. Such a model will be unable to capture the dependency that we see visually on the plot with the independent variable. The bias here, we say is high.

One way to remember this is to think, this model has a preconceived notion—or bias—that all vehicles have the same fuel efficiency. Now, if we fit a model with an intercept and a slope, then we have a model like the one

shown, this linear regression line. Here, the bias is lower than before. If we then fit a degree two polynomial, then we have a model, which is able to capture this roughly parabolic shape observed in the data. The bias is even lower. If we were to fit a degree 25 polynomial—not shown, since we only have 25 datapoints—our bias will go to zero. Our model would be able to perfectly fit the training data. Now, a very high degree model that has low or even zero bias.

But, it has a serious problem: Variance. We've mentioned variance before. Variance, roughly speaking, represents how sensitive the model is to the data. This model, the first one with only an intercept term, has low variance. Small changes to the data, result in only small changes to the model. Now by contrast, a higher-order model like this degree-6 model shown, they have high variance. So, even a small change to even a single data point, can result in a dramatically different model. So why am I bringing this up? Well, it's because bias and variance can be given formal mathematical definitions, though we will not do so in our course. Now, if you do find...formally define them, we can show that the expected loss—also known as the risk—is equal to the square of the bias plus the variance plus another term I won't describe.

So, as our model complexity increases, the bias decreases and the variance will tend to increase. So, visually, we see here the Error, the Bias^2 and the Variance of a model, versus the complexity in the classical regime. In this regime, our ideal model will sit at the sweet spot where the bias has gotten fairly low, but the variance has not yet taken off. By contrast, in the modern interpolating regime, the variance often increases, but then it begins to

decrease as we go deeper into the interpolating regime for reasons we alluded to earlier when discussing stochastic gradient descent.

This figure here comes from *Rethinking Bias–Variance Trade-Off for Generalization of Neural Networks*. That's a 2020 paper by Zitong Yang and other authors. And, in this paper, the authors experimentally compute the bias^2, the variance, and the expected loss for a model as a function of the model's complexity. The classical picture of the bias–variance tradeoff is shown at the leftmost part of this figure. Bias is decreasing, but variance is increasing. The sweet spot, representing the optimal model, is classically assumed to exist in this space.

However, for this specific experiment on neural networks: As we move into the interpolating regime, the variance begins to drop again, eventually reaching lower levels than for simpler models. At the same time, the bias continues to decrease as well. As a result, the optimal model is actually way out here towards the right-hand side of the figure. In this case, you might even say there's not really a bias–variance tradeoff. Instead, we might want to build our model as big as possible, as such a model minimizes both bias and variance.

And, in common practice today, many of the world's most advanced machine learning models are these absolutely gigantic neural networks with hundreds of billions of parameters or even more. And for such models, the number of parameters, it just vastly exceeds the amount of available data. Then, part of the reason that this all happens is the way that stochastic gradient descent chooses from among the infinite number of

models, with zero or very low training error. Note that we'll discuss neural networks themselves in a later module.

## Video 15: Conclusion

Let's reflect on what we've learned today. The star of our show was gradient descent, which allows us to find the minima of functions. How it works is that at each step of the procedure, we compute the steepest direction of the function that we're minimizing, giving us a p-dimensional vector, or gradient. Our next guess for the optimal solution is our current guess minus this p-dimensional vector, times the learning rate, alpha. Now for convex functions, the minimum that this gradient descent procedure finds, is guaranteed to be the global minimum. And there's the other benefit that optimization procedures on convex functions, they tend to converge much more quickly than on non-convex functions—even those that have a single minimum. Stochastic gradient descent lets us rapidly compute an approximation of the gradient. And we saw that it can be done in batches of data, or even on a single data point.

We also talked about how in classical machine learning, we try to pick a model complexity that minimizes the test error, that optimizes this bias–variance tradeoff. But in some modern machine learning settings, we pick the largest possible model as that minimizes both bias and variance. We saw that, empirically, we sometimes observe this double descent behavior. Whereas the number of parameters grows, we first see the test error increase, but then eventually decrease again. And understanding stochastic gradient descent provides some insight into why this counterintuitive phenomenon occurs. Alright, that's it for today. So, I hope you'll enjoy the homework and I'll see you next time.