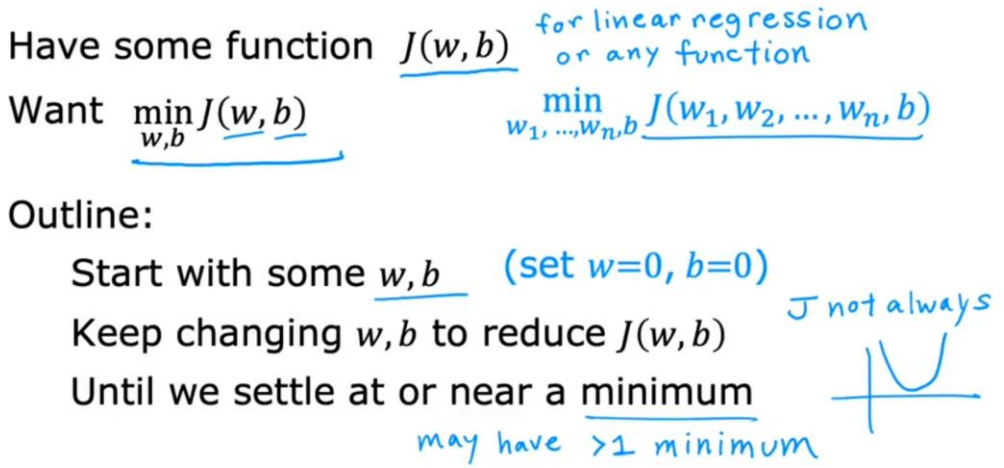
Descente de gradient

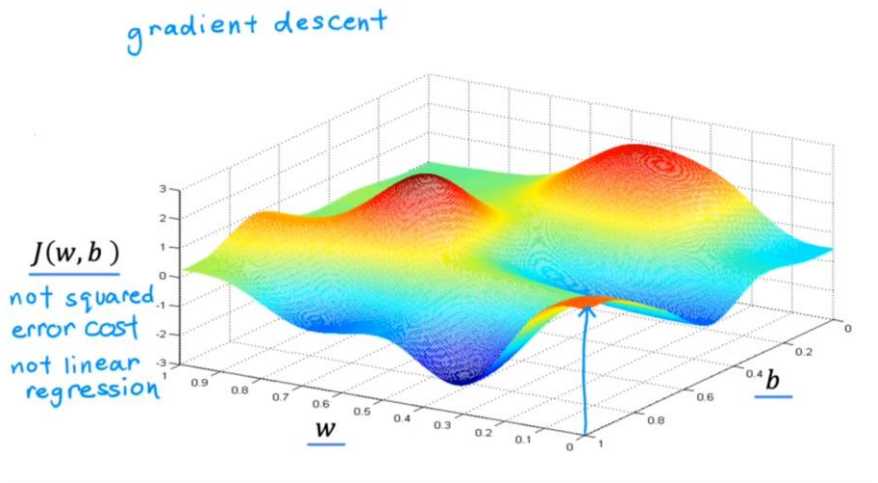
In the last video, we saw visualizations of the cost function j and how you can try different choices of the parameters w and b and see what cost value they get you. It would be nice if we had a more systematic way to find the values of w and b, that results in the smallest possible cost, j of w, b. It turns out there's an algorithm called gradient descent that you can use to do that. Gradient descent is used all over the place in machine learning, not just for linear regression, but for training for example some of the most advanced neural network models, also called deep learning models.



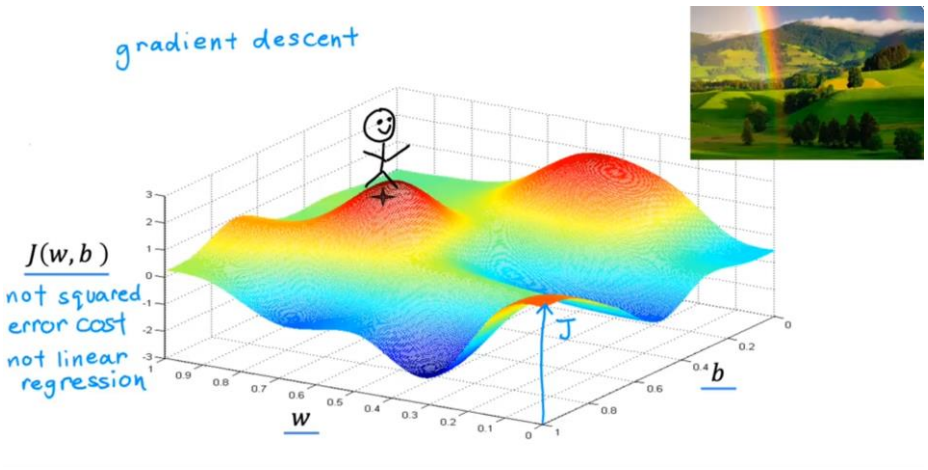
You have the cost function j of w, b right here that you want to minimize. In the example we've seen so far, this is a cost function for linear regression, but it turns out that gradient descent is an algorithm that you can use to try to minimize any function, not just a cost function for linear regression. Just to make this discussion on gradient descent more general, it turns out that gradient descent applies to more general functions, including other cost functions that work with models that have more than two parameters.

For instance, if you have a cost function J as a function of w_1, w_2 up to w_n and b, your objective is to minimize j over the parameters w_1 to w_n and b. In other words, you want to pick values for w_1 through w_n and b, that gives you the smallest possible value of j. It turns out that gradient descent is an algorithm that you can apply to try to minimize this cost function j as well. What you're going to do is just to start off with some initial guesses for w and b. In linear regression, it won't matter too much what the initial value are, so a common choice is to set them both to 0. For example, you can set w to 0 and b to 0 as the initial guess.

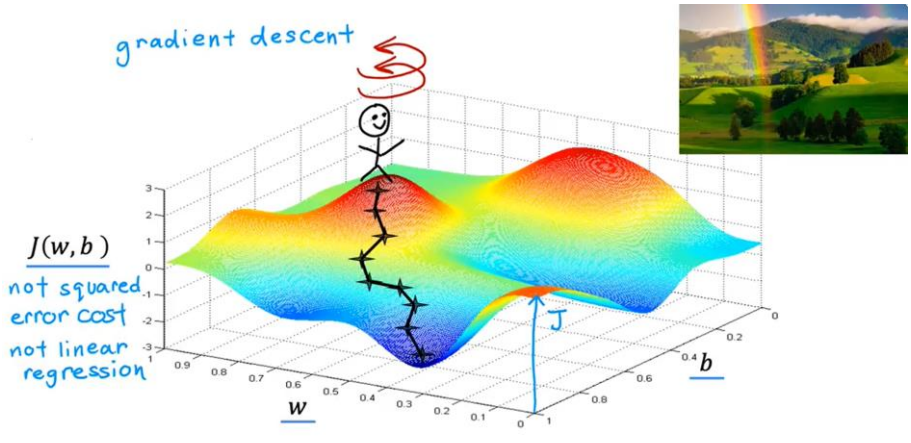
With the gradient descent algorithm, what you're going to do is, you'll keep on changing the parameters w and b a bit every time to try to reduce the cost j of w, b until hopefully j settles at or near a minimum. One thing I should note is that for some functions j that may not be a bow shape or a hammock shape, it is possible for there to be more than one possible minimum.



Let's take a look at an example of a more complex surface plot j to see what gradient is doing. This function is not a squared error cost function. For linear regression with the squared error cost function, you always end up with a bow shape or a hammock shape. But this is a type of cost function you might get if you're training a neural network model. Notice the axes, that is w and b on the bottom axis. For different values of w and b, you get different points on this surface, j of w, b, where the height of the surface at some point is the value of the cost function.



Now, let's imagine that this surface plot is actually a view of a slightly hilly outdoor park or a golf course where the high points are hills and the low points are valleys like so. I'd like you to imagine if you will, that you are physically standing at this point on the hill. If it helps you to relax, imagine that there's lots of really nice green grass and butterflies and flowers is a really nice hill.



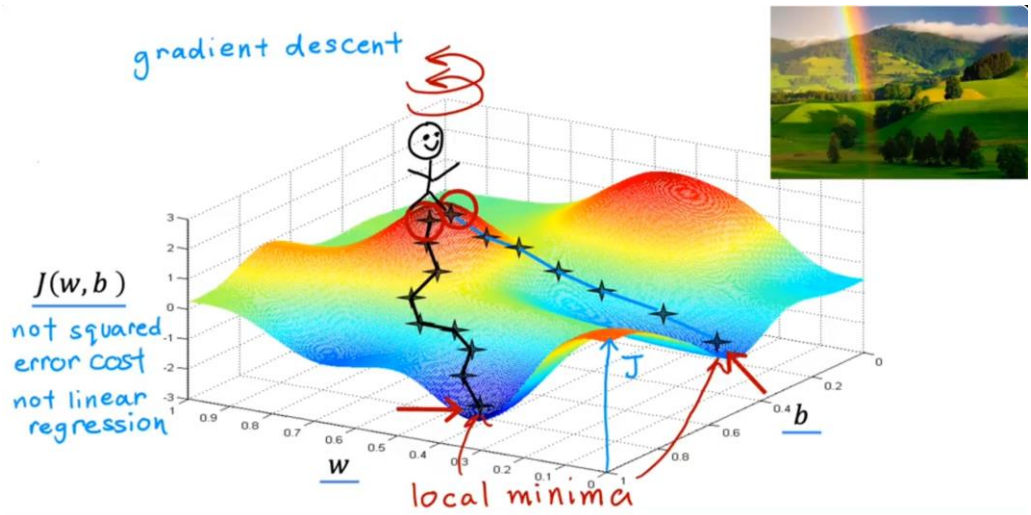
Your goal is to start up here and get to the bottom of one of these valleys as efficiently as possible. What the gradient descent algorithm does is, you're going to spin around 360 degrees and look around and ask yourself, if I were to take a tiny little baby step in one direction, and I want to go downhill as quickly as possible to or one of these valleys.

What direction do I choose to take that baby step? Well, if you want to walk down this hill as efficiently as possible, it turns out that if you're standing at this point in the hill and you look around, you will notice that the best direction to take your next step downhill is roughly that direction.

Mathematically, this is the direction of steepest descent. It means that when you take a tiny baby little step, this takes you downhill faster than a tiny little baby step you could have taken in any other direction. After taking this first step, you're now at this point on the hill over here.

Now let's repeat the process. Standing at this new point, you're going to again spin around 360 degrees and ask yourself, in what direction will I take the next little baby step in order to move downhill? If you do that and take another step, you end up moving a bit in that direction and you can keep going.

From this new point, you can again look around and decide what direction would take you downhill most quickly. Take another step, another step, and so on, until you find yourself at the bottom of this valley, at this local minimum, right here.



What you just did was go through multiple steps of gradient descent. It turns out, gradient descent has an interesting property. Remember that you can choose a starting point at the surface by choosing starting values for the parameters w and b. When you perform gradient descent a moment ago, you had started at this point over here.

Now, imagine if you try gradient descent again, but this time you choose a different starting point by choosing parameters that place your starting point just a couple of steps to the right over here. If you then repeat the gradient descent process, which means you look around, take a little step in the direction of steepest ascent so you end up here.

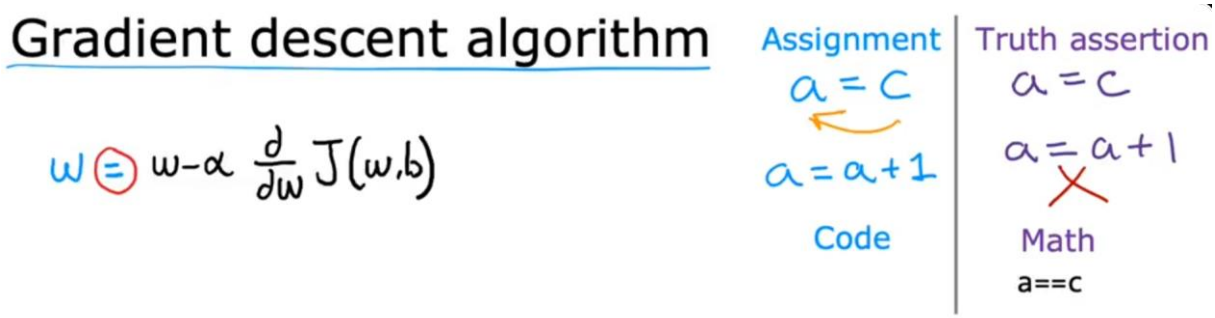
Then you again look around, take another step, and so on. If you were to run gradient descent this second time, starting just a couple steps in the right of where we did it the first time, then you end up in a totally different valley. This different minimum over here on the right.

The bottoms of both the first and the second valleys are called local minima. Because if you start going down the first valley, gradient descent won't lead you to the second valley, and the same is true if you

started going down the second valley, you stay in that second minimum and not find your way into the first local minimum. This is an interesting property of the gradient descent algorithm, and you see more about this later.

Mise en œuvre de la descente de gradient

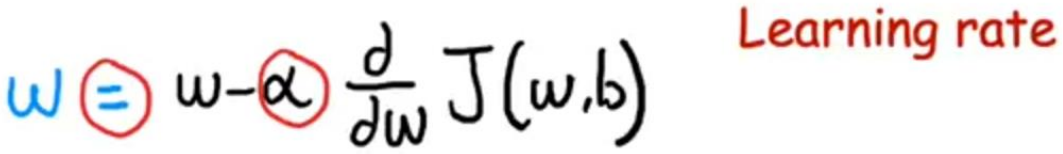
Let's take a look at how you can actually implement the gradient descent algorithm. Let me write down the gradient descent algorithm. Here it is. On each step, w, the parameter, is updated to the old value of w minus Alpha times this term d/dw of the cos function J of wb. What this expression is saying is, after your parameter w by taking the current value of w and adjusting it a small amount, which is this expression on the right, minus Alpha times this term over here.



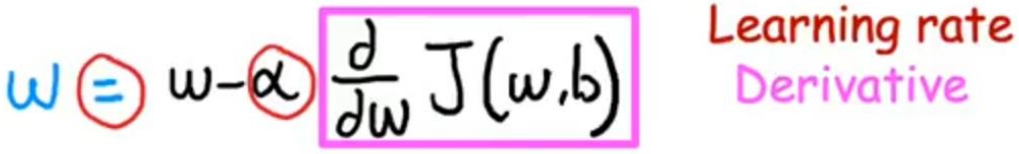
First, this equal notation here. Now, since I said we're assigning w a value using this equal sign, so in this context, this equal sign is the assignment operator. Specifically, in this context, if you write code that says a equals c, it means take the value c and store it in your computer, in the variable a. Or if you write a equals a plus 1, it means set the value of a to be equal to a plus 1, or increments the value of a by one. The assignment operator encoding is different than truth assertions in mathematics.

Where if I write a equals c, I'm asserting, that is, I'm claiming that the values of a and c are equal to each other. Hopefully, I will never write a truth assertion a equals a plus 1 because that just can't possibly be true. In Python and in other programming languages, truth assertions are sometimes written as equals equals, so you may see oh, that says a equals equals c if you're testing whether a is equal to c. But in math notation, as we conventionally use it, like in these videos, the equal sign can be used for either assignments or for truth assertion.

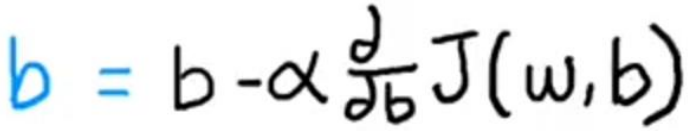
I try to make sure I was clear when I write an equal sign, whether we're assigning a value to a variable, or whether we're asserting the truth of the equality of two values.



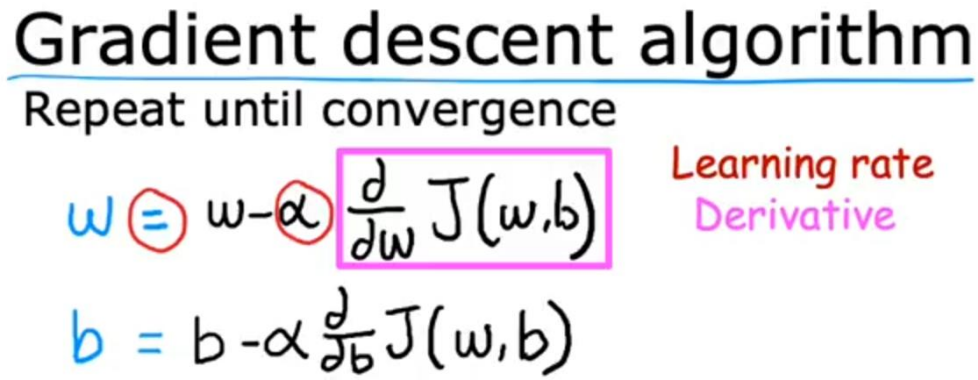
The symbol here is the Greek alphabet Alpha. In this equation, Alpha is also called the learning rate. The learning rate is usually a small positive number between 0 and 1 and it might be say, 0.01. What Alpha does is, it basically controls how big of a step you take downhill. If Alpha is very large, then that corresponds to a very aggressive gradient descent procedure where you're trying to take huge steps downhill. If Alpha is very small, then you'd be taking small baby steps downhill.



Finally, this term here, that's the derivative term of the cost function J. Let's not worry about the details of this derivative right now. But later on, you'll get to see more about the derivative term. But for now, you can think of this derivative term that I drew a magenta box around as telling you in which direction you want to take your baby step. In combination with the learning rate Alpha, it also determines the size of the steps you want to take downhill. Now, I do want to mention that derivatives come from calculus. Even if you aren't familiar with calculus, don't worry about it. Even without knowing any calculus, you'd be able to figure out all you need to know about this derivative term in this video and the next. One more thing.



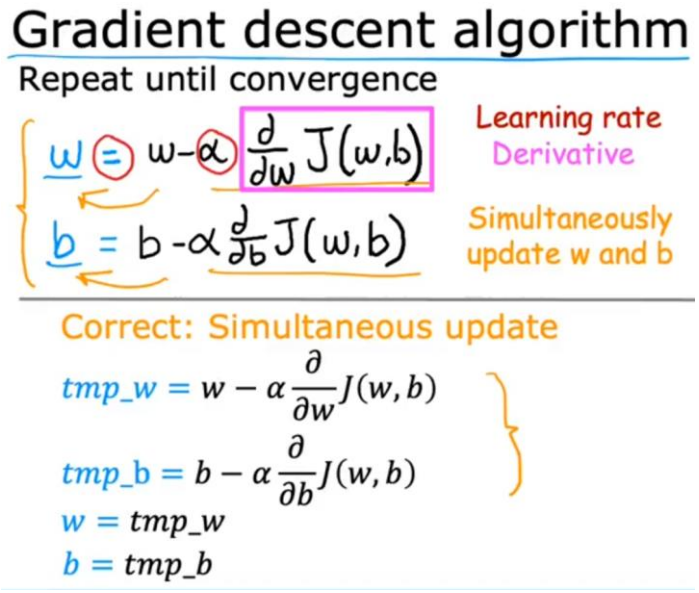
Remember your model has two parameters, not just w, but also b. You also have an assignment operations update the parameter b that looks very similar. b is assigned the old value of b minus the learning rate Alpha times this slightly different derivative term, d/db of J of wb.



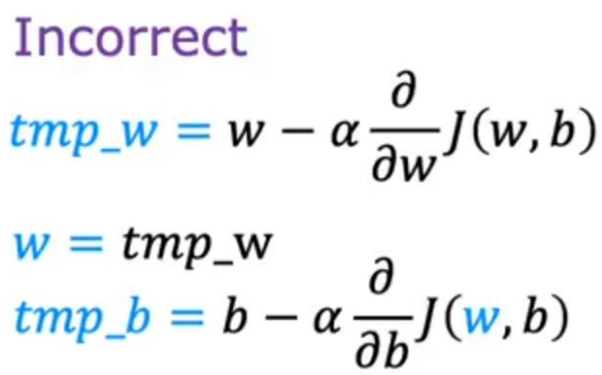
Remember in the graph of the surface plot where you're taking baby steps until you get to the bottom of the value, well, for the gradient descent algorithm, you're going to repeat these two update steps until the algorithm converges. By converges, I mean that you reach the point at a local minimum where the parameters w and b no longer change much with each additional step that you take.

Now, there's one more subtle detail about how to correctly in semantic gradient descent, you're going to update two parameters, w and b. This update takes place for both parameters, w and b. One important detail is that for gradient descent, you want to simultaneously update w and b, meaning you want to update both parameters at the same time. What I mean by that, is that in this expression, you're going to update w from the old w to a new w, and you're also updating b from its oldest value to a new value of b. The way to implement this is to compute the right side, computing this thing for w

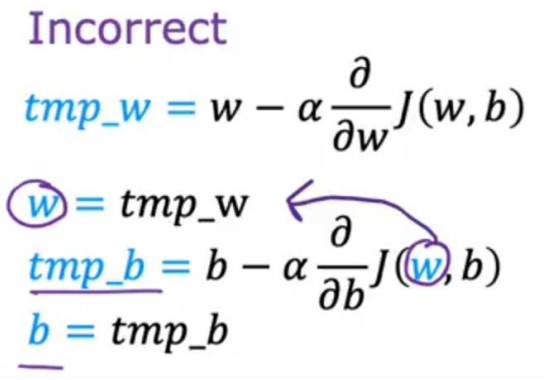
and b, and simultaneously at the same time, update w and b to the new values. Let's take a look at what this means.



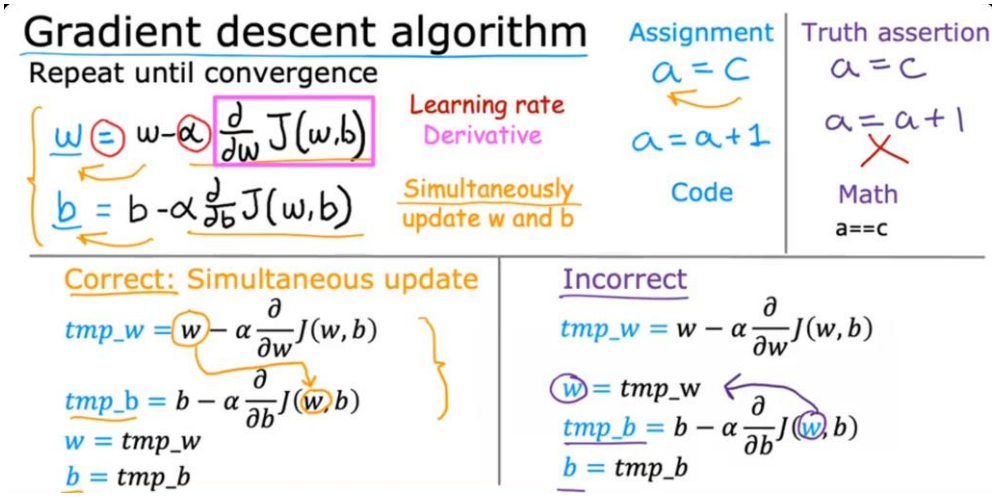
Here's the correct way to implement gradient descent which does a simultaneous update. This sets a variable temp_w equal to that expression, which is w minus that term here. There's also a set in another variable temp_b to that, which is b minus that term. You compute both for hand sides, both updates, and store them into variables temp_w and temp_b. Then you copy the value of temp_w into w, and you also copy the value of temp_b into b.



Now, one thing you may notice is that this value of w is from the for w gets updated. Here, I noticed that the pre-update w is where it goes into the derivative term over here. In contrast, here is an incorrect implementation of gradient descent that does not do a simultaneous update. In this incorrect implementation, we compute temp_w, same as before, so far that's okay. Now here's where things start to differ. We then update w with the value in temp_w before calculating the new value for the other parameter to be. Next, we calculate temp_b as b minus that term here, and finally, we update b with the value in temp_b.



The difference between the right-hand side and the left-hand side implementations is that if you look over here, this w has already been updated to this new value, and this is updated w that actually goes into the cost function j of w, b. It means that this term here on the right is not the same as this term over here that you see on the left. That also means this temp_b term on the right is not quite the same as the temp b term on the left, and thus this updated value for b on the right is not the same as this updated value for variable b on the left. The way that gradient descent is implemented in code, it actually turns out to be more natural to implement it the correct way with simultaneous updates. When you hear someone talk about gradient descent, they always mean the gradient descents where you perform a simultaneous update of the parameters.



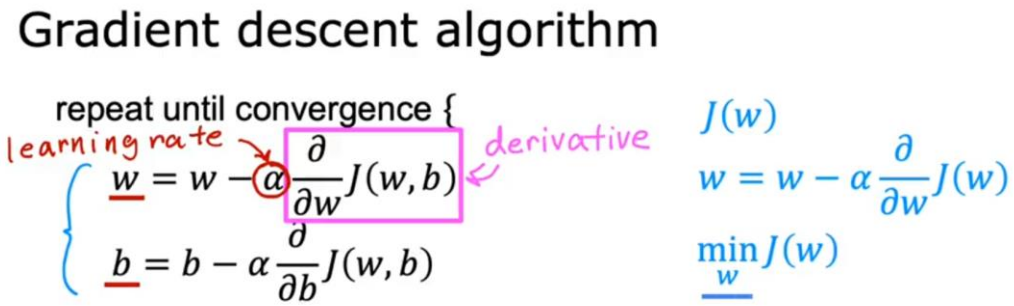
If however, you were to implement non-simultaneous update, it turns out it will probably work more or less anyway. But doing it this way isn't really the correct way to implement it, is actually some other algorithm with different properties. I would advise you to just stick to the correct simultaneous update and not use this incorrect version on the right. That's gradient descent.

L'intuition de la descente de gradient

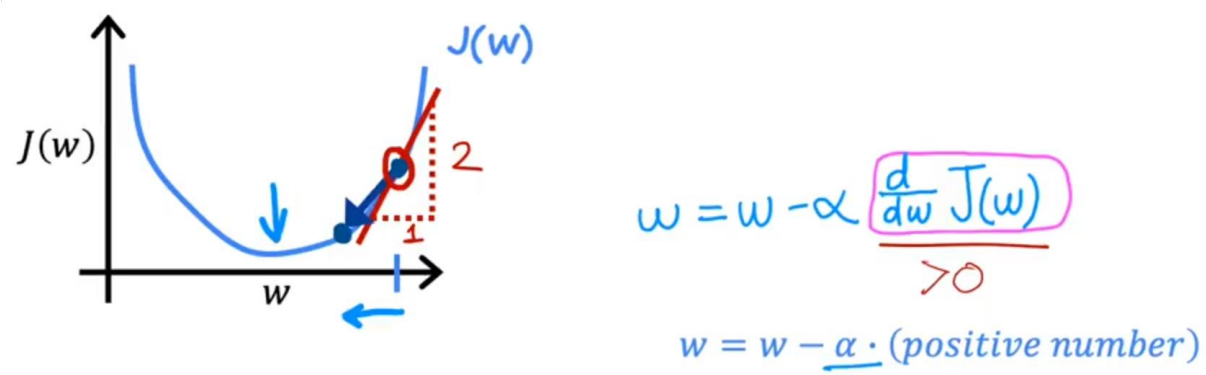
Here's the gradient descent algorithm that you saw in the previous video. As a reminder, this variable, this Greek symbol Alpha, is the learning rate. The learning rate controls how big of a step you take when updating the model's parameters, w and b. This term here, this d over dw, this is a derivative term. By convention in math, this d is written with this funny font here.

What we're going to focus on now is get more intuition about what this learning rate and what this derivative are doing and why when multiplied together like this, it results in updates to parameters w

and b. That makes sense. In order to do this let's use a slightly simpler example where we work on minimizing just one parameter. Let's say that you have a cost function J of just one parameter w with w is a number. This means the gradient descent now looks like this.



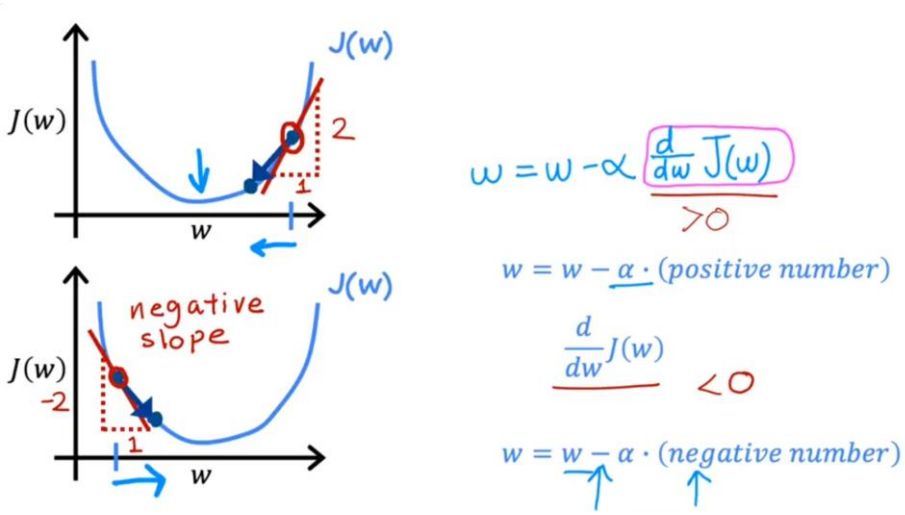
W is updated to w minus the learning rate Alpha times d over dw of J of w. You're trying to minimize the cost by adjusting the parameter w. This is like our previous example where we had temporarily set b equal to 0 with one parameter w instead of two, you can look at two-dimensional graphs of the cost function j, instead of three dimensional graphs.



Let's look at what gradient descent does on just function J of w. Here on the horizontal axis is parameter w, and on the vertical axis is the cost j of w. Now less initialized gradient descent with some starting value for w. Let's initialize it at this location. Imagine that you start off at this point right here on the function J, what gradient descent will do is it will update w to be w minus learning rate Alpha times d over dw of J of w. Let's look at what this derivative term here means.

A way to think about the derivative at this point on the line is to draw a tangent line, which is a straight line that touches this curve at that point. Enough, the slope of this line is the derivative of the function j at this point. To get the slope, you can draw a little triangle like this. If you compute the height divided by the width of this triangle, that is the slope. For example, this slope might be 2 over 1, for instance and when the tangent line is pointing up and to the right, the slope is positive, which means that this derivative is a positive number, so is greater than 0.

The updated w is going to be w minus the learning rate times some positive number. The learning rate is always a positive number. If you take w minus a positive number, you end up with a new value for w, that's smaller. On the graph, you're moving to the left, you're decreasing the value of w. You may notice that this is the right thing to do if your goal is to decrease the cost J, because when we move towards the left on this curve, the cost j decreases, and you're getting closer to the minimum for J, which is over here. So far, gradient descent, seems to be doing the right thing.



Now, let's look at another example. Let's take the same function j of w as above, and now let's say that you initialized gradient descent at a different location. Say by choosing a starting value for w that's over here on the left. That's this point of the function j. Now, the derivative term, remember is d over dw of J of w, and when we look at the tangent line at this point over here, the slope of this line is a derivative of J at this point. But this tangent line is sloping down into the right.

This lines sloping down into the right has a negative slope. In other words, the derivative of J at this point is a negative number. For instance, if you draw a triangle, then the height like this is negative 2 and the width is 1, the slope is negative 2 divided by 1, which is negative 2, which is a negative number. When you update w, you get w minus the learning rate times a negative number. This means you subtract from w, a negative number. But subtracting a negative number means adding a positive number, and so you end up increasing w.

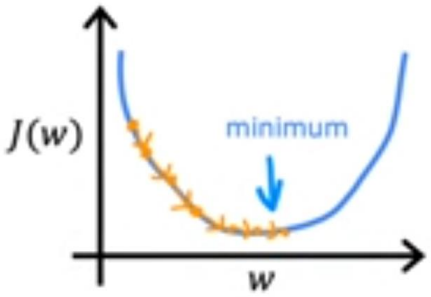
Because subtracting a negative number is the same as adding a positive number to w. This step of gradient descent causes w to increase, which means you're moving to the right of the graph and your cost J has decrease down to here. Again, it looks like gradient descent is doing something reasonable, is getting you closer to the minimum.

Taux d'apprentissage

The choice of the learning rate, alpha will have a huge impact on the efficiency of your implementation of gradient descent. And if alpha, the learning rate is chosen poorly rate of descent may not even work at all. In this video, let's take a deeper look at the learning rate. This will also help you choose better learning rates for your implementations of gradient descent. So here again, is the gradient descent rule. W is updated to be W minus the learning rate, alpha times the derivative term. To learn more about what the learning rate alpha is doing. Let's see what could happen if the learning rate alpha is either too small or if it is too large.

$$w = w - \alpha \frac{d}{dw} J(w)$$

If α is too small...
Gradient descent may be slow.



For the case where the learning rate is too small. Here's a graph where the horizontal axis is W and the vertical axis is the cost J. And here's the graph of the function J of W. Let's start grading descent at this point here, if the learning rate is too small. Then what happens is that you multiply your derivative term by some really, really small number.

So you're going to be multiplying by number alpha. That's really small, like 0.0000001. And so you end up taking a very small baby step like that. Then from this point you're going to take another tiny tiny little baby step. But because the learning rate is so small, the second step is also just minuscule. The outcome of this process is that you do end up decreasing the cost J but incredibly slowly.

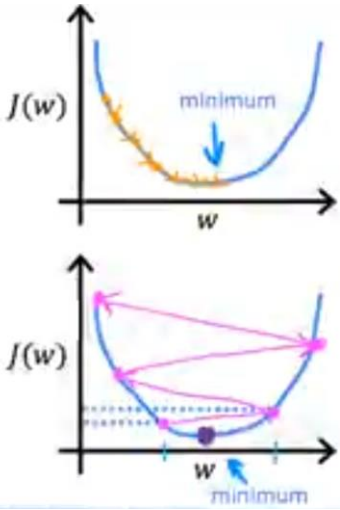
So, here's another step and another step, another tiny step until you finally approach the minimum. But as you may notice you're going to need a lot of steps to get to the minimum. So to summarize if the learning rate is too small, then gradient descents will work, but it will be slow. It will take a very long time because it's going to take these tiny tiny baby steps. And it's going to need a lot of steps before it gets anywhere close to the minimum.

$$w = w - \alpha \frac{d}{dw} J(w)$$

If α is too small...
Gradient descent may be slow.

If α is too large...

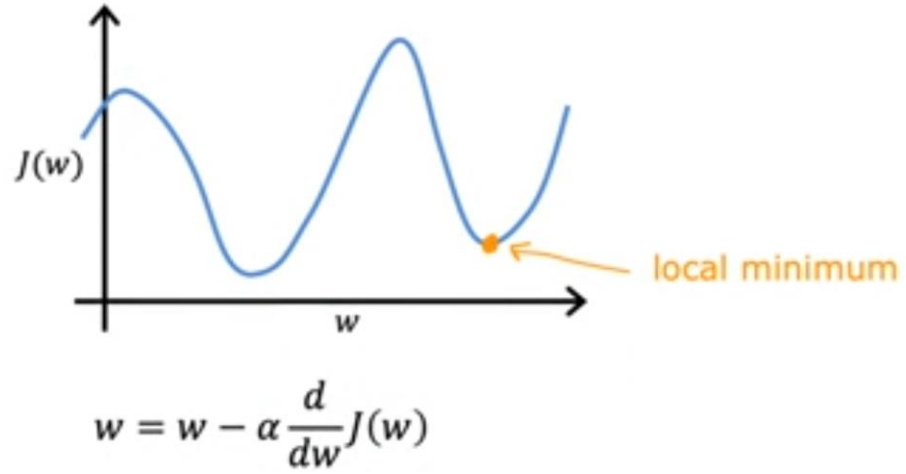
Gradient descent may:
- Overshoot, never reach minimum
- Fail to converge, diverge



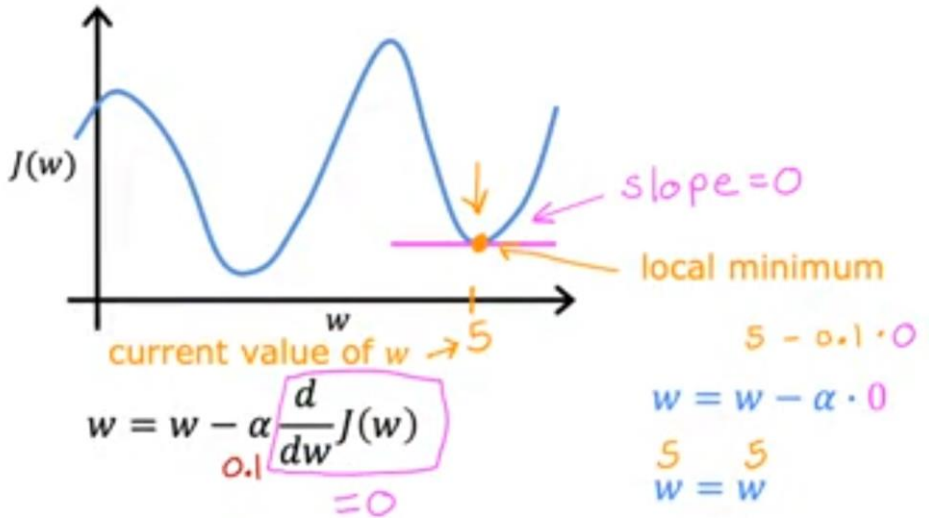
Now, let's look at a different case. What happens if the learning rate is too large? Here's another graph of the cost function. And let's say we start grating descent with W at this value here. So it's actually already pretty close to the minimum. So the decorative points to the right. But if the learning rate is too large then you update W very giant step to be all the way over here. And that's this point here on the function J.

So you move from this point on the left, all the way to this point on the right. And now the cost has actually gotten worse. It has increased because it started out at this value here and after one step, it actually increased to this value here. Now the derivative at this new point says to decrease W but when the learning rate is too big. Then you may take a huge step going from here all the way out here. So now you've gotten to this point here and again, if the learning rate is too big.

Then you take another huge step with an acceleration and way overshoot the minimum again. So now you're at this point on the right and one more time you do another update. And end up all the way here and so you're now at this point here. So as you may notice you're actually getting further and further away from the minimum. So if the learning rate is too large, then creating the sense may overshoot and may never reach the minimum. And another way to say that is that great intersect may fail to converge and may even diverge.



So, here's another question, you may be wondering one of your parameter W is already at this point here. So that your cost J is already at a local minimum. What do you think? One step of gradient descent will do if you've already reached a minimum? So this is a tricky one.

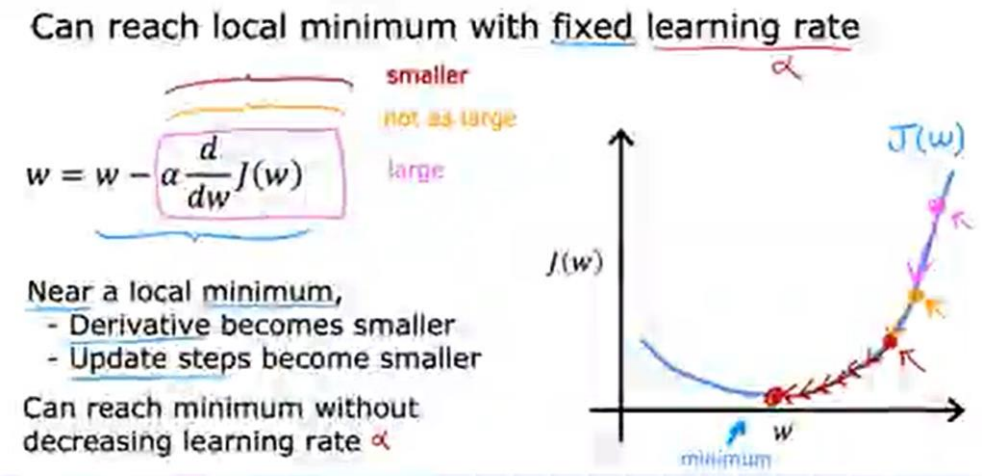


But let's work through this together. Let's suppose you have some cost function J. And the one you see here isn't a square error cost function and this cost function has two local minima corresponding to the two valleys that you see here. Now let's suppose that after some number of steps of gradient descent, your parameter W is over here, say equal to five. And so this is the current value of W. This means that you're at this point on the cost function J. And that happens to be a local minimum, turns out if you draw attention to the function at this point.

The slope of this line is zero and thus the derivative term. Here is equal to zero for the current value of W. And so you're grading descent update becomes W is updated to W minus the learning rate times

zero. We're here that's because the derivative term is equal to zero. And this is the same as saying let's set W to be equal to W. So this means that if you're already at a local minimum, gradient descent leaves W unchanged. Because it just updates the new value of W to be the exact same old value of W.

So concretely, let's say if the current value of W is five. And alpha is 0.1 after one iteration, you update W as W minus alpha times zero and it is still equal to five. So if your parameters have already brought you to a local minimum, then further gradient descent steps to absolutely nothing. It doesn't change the parameters which is what you want because it keeps the solution at that local minimum.



This also explains why gradient descent can reach a local minimum, even with a fixed learning rate alpha. Here's what I mean, to illustrate this, let's look at another example. Here's the cost function J of W that we want to minimize. Let's initialize gradient descent up here at this point. If we take one update step, maybe it will take us to that point. And because this derivative is pretty large, grading, descent takes a relatively big step right.

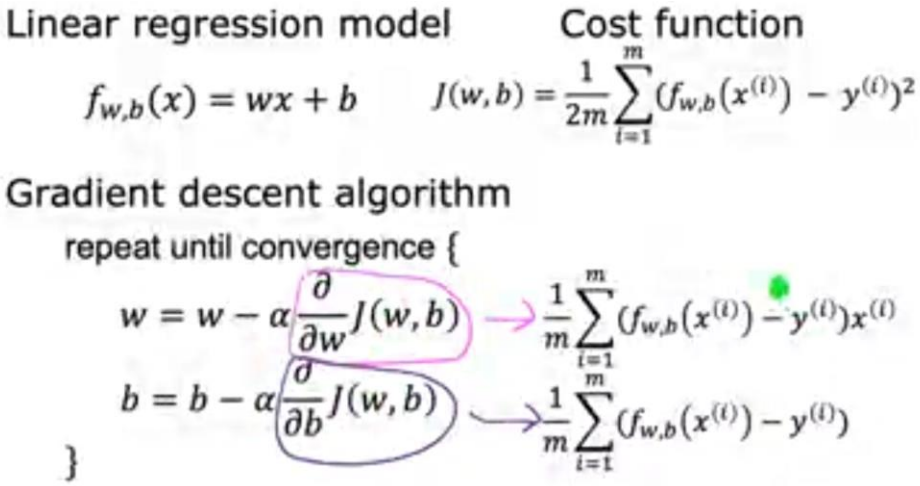
Now, we're at this second point where we take another step. And you may notice that the slope is not as steep as it was at the first point. So the derivative isn't as large. And so the next update step will not be as large as that first step. Now, read this third point here and the derivative is smaller than it was at the previous step. And will take an even smaller step as we approach the minimum. The decorative gets closer and closer to zero.

So as we run gradient descent, eventually we're taking very small steps until you finally reach a local minimum. So just to recap, as we get nearer a local minimum gradient descent will automatically take smaller steps. And that's because as we approach the local minimum, the derivative automatically gets smaller. And that means the update steps also automatically gets smaller. Even if the learning rate alpha is kept at some fixed value. So that's the gradient descent algorithm, you can use it to try to minimize any cost function J. Not just the mean squared error cost function that we're using for the new regression.

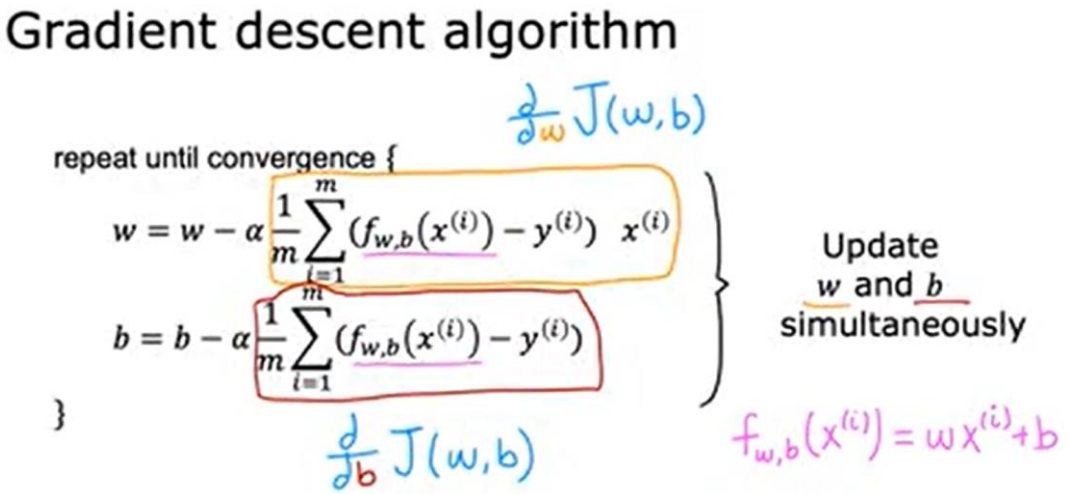
Descente de gradient pour la régression linéaire

Let's get to it. Here's the linear regression model. To the right is the squared error cost function. Below is the gradient descent algorithm. It turns out if you calculate these derivatives, these are the terms you would get. The derivative with respect to W is this 1 over m, sum of i equals 1 through m. Then the error term, that is the difference between the predicted and the actual values times the input feature

xi. The derivative with respect to b is this formula over here, which looks the same as the equation above, except that it doesn't have that xi term at the end. If you use these formulas to compute these two derivatives and implements gradient descent this way, it will work.

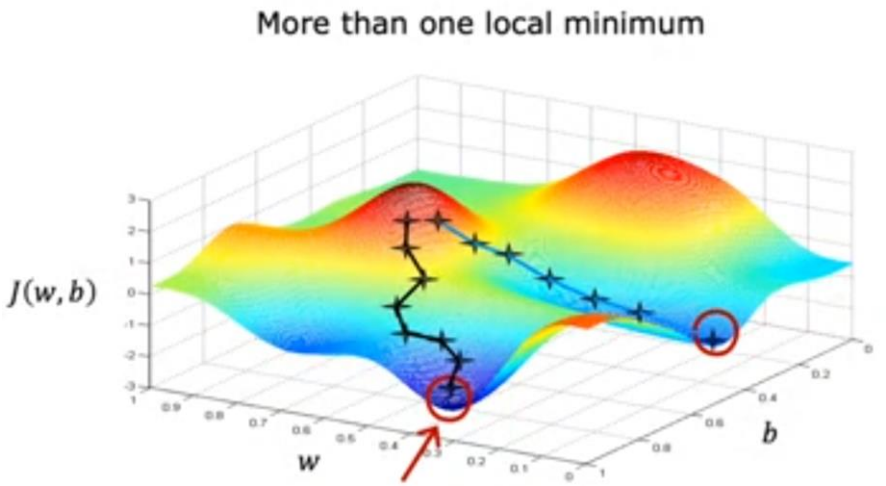


Now you have these two expressions for the derivatives. You can plug them into the gradient descent algorithm. Here's the gradient descent algorithm for linear regression. You repeatedly carry out these updates to w and b until convergence. Remember that this f of x is a linear regression model, so as equal to w times x plus b. This expression here is the derivative of the cost function with respect to w. This expression is the derivative of the cost function with respect to b. Just as a reminder, you want to update w and b simultaneously on each step.

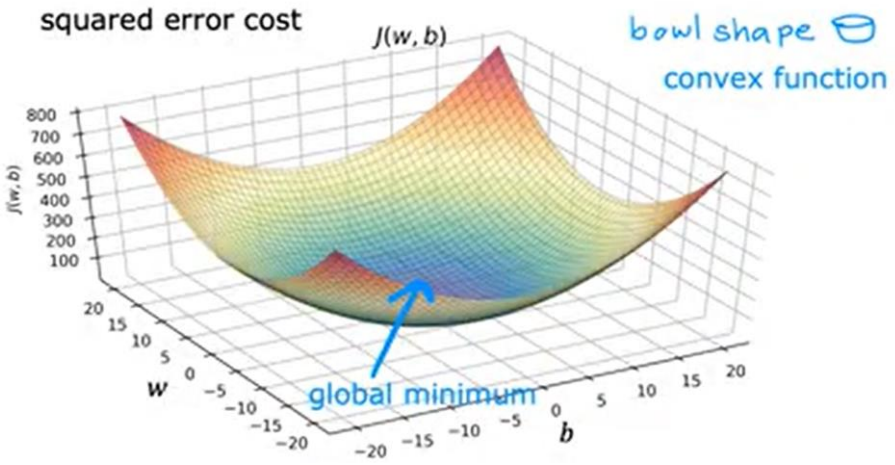


Now, let's get familiar with how gradient descent works. One the shoe we saw with gradient descent is that it can lead to a local minimum instead of a global minimum. Whether global minimum means the point that has the lowest possible value for the cost function J of all possible points.

You may recall this surface plot that looks like an outdoor park with a few hills with the process and the birds as a relaxing Hobo Hill. This function has more than one local minimum. Remember, depending on where you initialize the parameters w and b, you can end up at different local minima. You can end up here, or you can end up here.



But it turns out when you're using a squared error cost function with linear regression, the cost function does not and will never have multiple local minima. It has a single global minimum because of this bowl-shape. The technical term for this is that this cost function is a convex function. Informally, a convex function is of bowl-shaped function and it cannot have any local minima other than the single global minimum. When you implement gradient descent on a convex function, one nice property is that so long as you're learning rate is chosen appropriately, it will always converge to the global minimum.



Exécution de la descente de gradient

Let's go see the algorithm in action. Here's a plot of the model and data on the upper left and a contour plot of the cost function on the upper right and at the bottom is the surface plot of the same cost function. Often w and b will both be initialized to 0, but for this demonstration, lets initialized w = -0.1 and b = 900. So this corresponds to f(x) = -0.1x + 900.

Now, if we take one step using gradient descent, we ended up going from this point of the cost function out here to this point just down and to the right and notice that the straight line fit is also changed a bit.

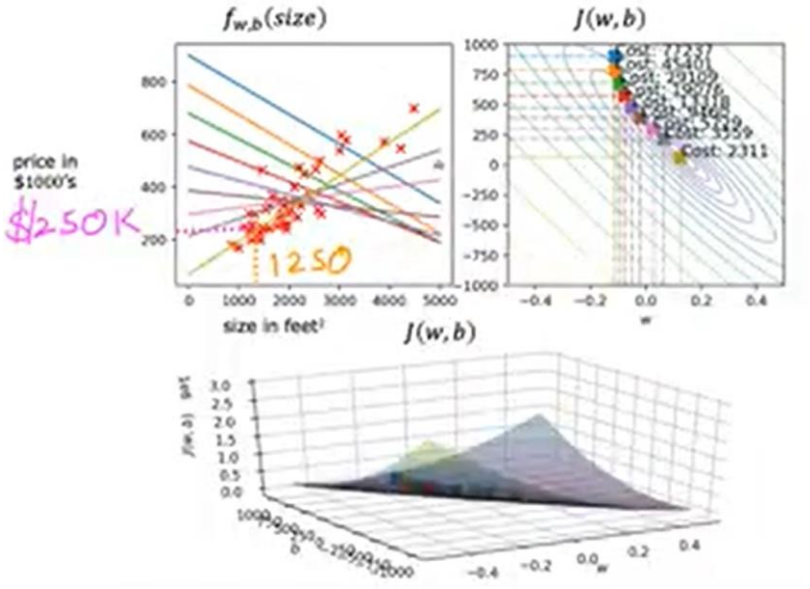
Let's take another step.

The cost function has now moved to this third and again the function f(x) has also changed a bit.

As you take more of these steps, the cost is decreasing at each update. So the parameters w and b are following this trajectory

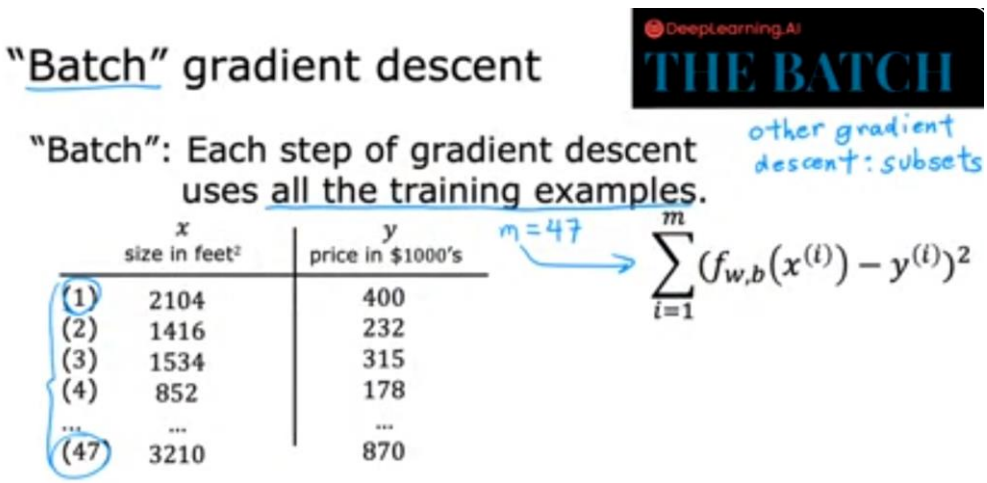
And if you look on the left, you get this corresponding straight line fit that fits the data better and better until we've reached the global minimum. The global minimum corresponds to this straight line fit, which is a relatively good fit to the data. I mean, isn't that cool.

And so that's gradient descent and we're going to use this to fit a model to the holding data. And you can now use this f(x) model to predict the price of your clients house or anyone else's house. For instance, if your friend's house size is 1250 square feet, you can now read off the value and predict that maybe they could get, I don't know, $250,000 for the house.



To be more precise, this gradient descent process is called batch gradient descent. The term batch gradient descent refers to the fact that on every step of gradient descent, we're looking at all of the training examples, instead of just a subset of the training data.

So in computing grading descent, when computing derivatives, when computing the sum from i =1 to m. And bash gradient descent is looking at the entire batch of training examples at each update. I know that bash grading percent may not be the most intuitive name, but this is what people in the machine learning community call it. If you've heard of the newsletter The Batch, that's published by DeepLearning.AI. The newsletter The batch was also named for this concept in machine learning.



And then it turns out that there are other versions of gradient descent that do not look at the entire training set, but instead looks at smaller subsets of the training data at each update step. But we'll use batch gradient descent for linear regression.

So that's it for linear regression.