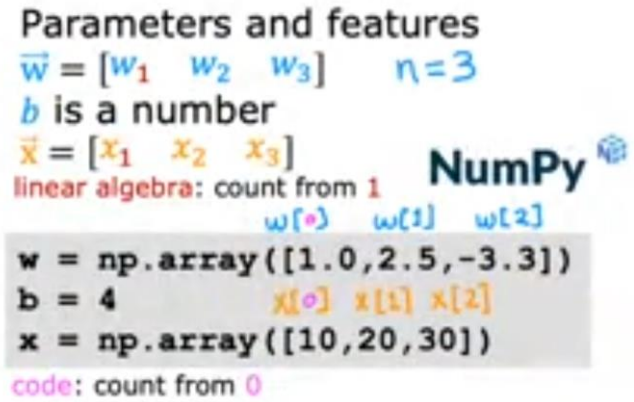
Vectorisation partie 1

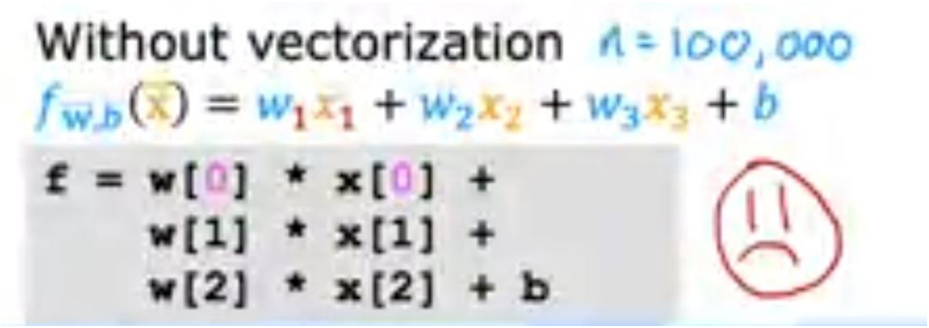
In this video, you see a very useful idea called vectorization. When you're implementing a learning algorithm, using vectorization will both make your code shorter and also make it run much more efficiently. Learning how to write vectorized code will allow you to also take advantage of modern numerical linear algebra libraries, as well as maybe even GPU hardware that stands for graphics processing unit. This is hardware objectively designed to speed up computer graphics in your computer, but turns out can be used when you write vectorized code to also help you execute your code much more quickly.



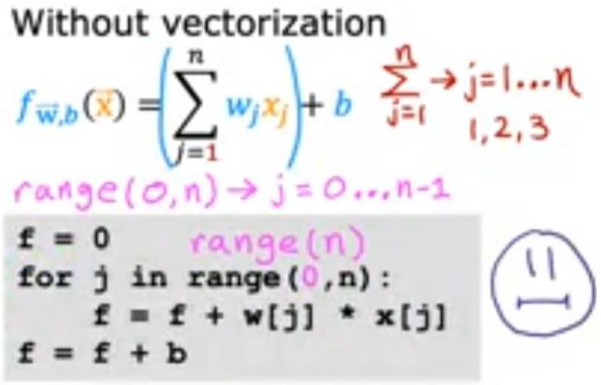
Here's an example with parameters w and b, where w is a vector with three numbers, and you also have a vector of features x with also three numbers. Here n is equal to 3. Notice that in linear algebra, the index or the counting starts from 1 and so the first value is subscripted w1 and x1. In Python code, you can define these variables w, b, and x using arrays like this.

Here, I'm actually using a numerical linear algebra library in Python called NumPy, which is by far the most widely used numerical linear algebra library in Python and in machine learning. Because in Python, the indexing of arrays while counting in arrays starts from 0, you would access the first value of w using w square brackets 0.

The second value using w square bracket 1, and the third and using w square bracket 2. The indexing here, it goes from 0,1 to 2 rather than 1, 2 to 3. Similarly, to access individual features of x, you will use x0, x1, and x2. Many programming languages including Python start counting from 0 rather than 1.

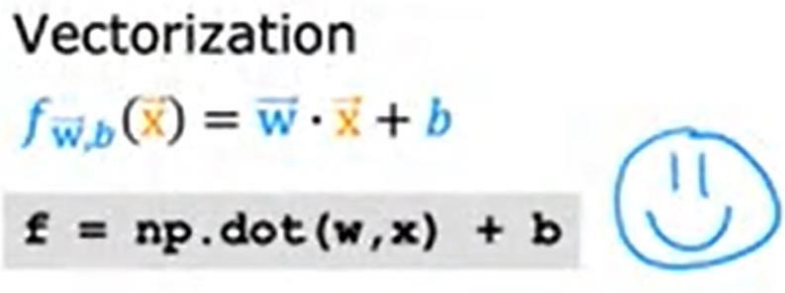


Now, let's look at an implementation without vectorization for computing the model's prediction. In codes, it will look like this. You take each parameter w and multiply it by his associated feature. Now, you could write your code like this, but what if n isn't three but instead n is a 100 or a 100,000 is both inefficient for you the code and inefficient for your computer to compute.



Here's another way. Without using vectorization but using a for loop. In math, you can use a summation operator to add all the products of w_j and x_j for j equals 1 through n. Then I'll cite the summation you add b at the end. To summation goes from j equals 1 up to and including n. For n equals 3, j therefore goes from 1, 2 to 3. In code, you can initialize after 0. Then for j in range from 0 to n, this actually makes j go from 0 to n minus 1.

From 0, 1 to 2, you can then add to f the product of w_j times x_j. Finally, outside the for loop, you add b. Notice that in Python, the range 0 to n means that j goes from 0 all the way to n minus 1 and does not include n itself. This is written range n in Python. But in this video, I added a 0 here just to emphasize that it starts from 0. While this implementation is a bit better than the first one, this still doesn't use factorization, and isn't that efficient?



Now, let's look at how you can do this using vectorization. This is the math expression of the function f, which is the dot product of w and x plus b, and now you can implement this with a single line of code by computing fp equals np dot dot, I said dot dot because the first dot is the period and the second dot is the function or the method called DOT. But is fp equals np dot dot w comma x and this implements the mathematical dot products between the vectors w and x. Then finally, you can add b to it at the end.

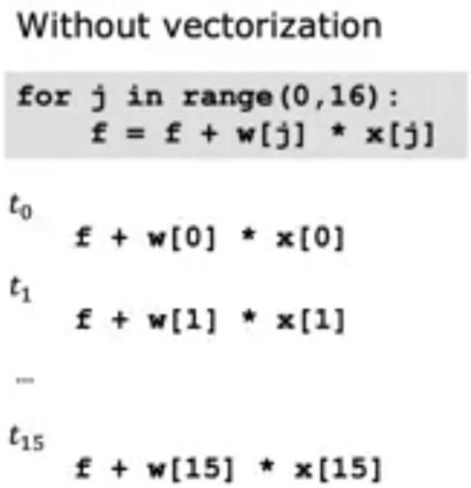
This NumPy dot function is a vectorized implementation of the dot product operation between two vectors and especially when n is large, this will run much faster than the two previous code examples. I want to emphasize that vectorization actually has two distinct benefits. First, it makes code shorter, is now just one line of code. Isn't that cool? Second, it also results in your code running much faster than either of the two previous implementations that did not use vectorization. The reason that the vectorized implementation is much faster is behind the scenes.

The NumPy dot function is able to use parallel hardware in your computer and this is true whether you're running this on a normal computer, that is on a normal computer CPU or if you are using a GPU, a graphics processor unit, that's often used to accelerate machine learning jobs. The ability of the NumPy dot function to use parallel hardware makes it much more efficient than the for loop or the sequential calculation that we saw previously.

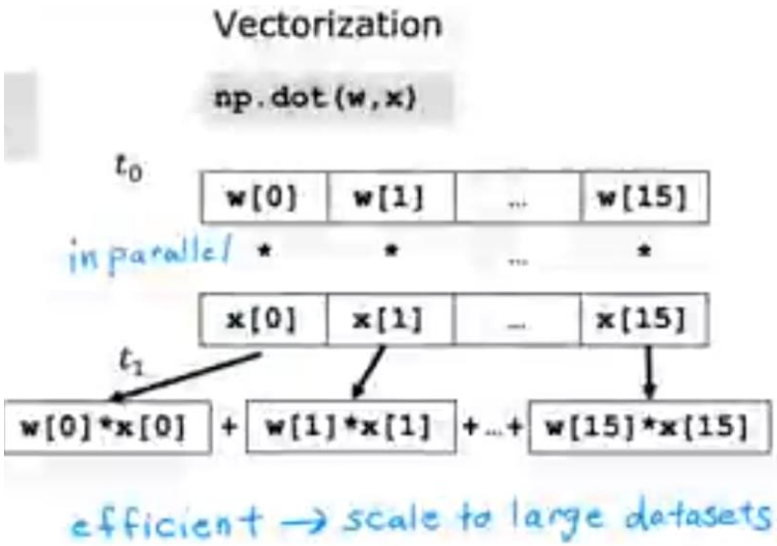
Now, this version is much more practical when n is large because you are not typing w0 times x0 plus w1 times x1 plus lots of additional terms like you would have had for the previous version. But while this saves a lot on the typing, is still not that computationally efficient because it still doesn't use vectorization.

Vectorisation partie 2

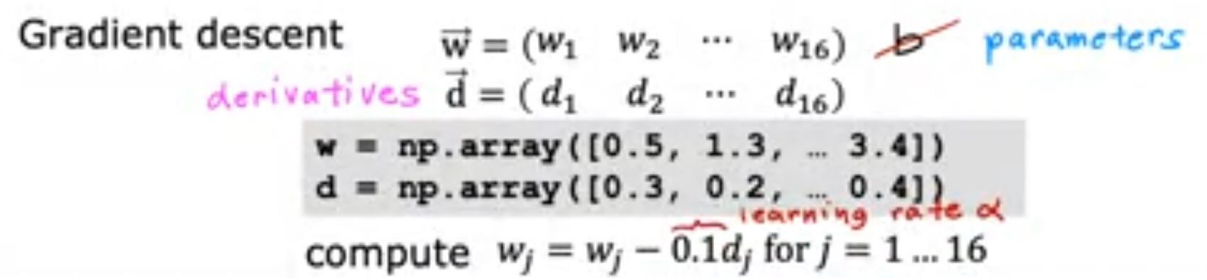
Let's look at this for loop. The for loop like this runs without vectorization. If j ranges from 0 to say 15, this piece of code performs operations one after another. On the first timestamp which I'm going to write as t0. It first operates on the values at index 0. At the next time-step, it calculates values corresponding to index 1 and so on until the 15th step, where it computes that. In other words, it calculates these computations one step at a time, one step after another.



In contrast, this function in NumPy is implemented in the computer hardware with vectorization. The computer can get all values of the vectors w and x, and in a single-step, it multiplies each pair of w and x with each other all at the same time in parallel. Then after that, the computer takes these 16 numbers and uses specialized hardware to add them altogether very efficiently, rather than needing to carry out distinct additions one after another to add up these 16 numbers.

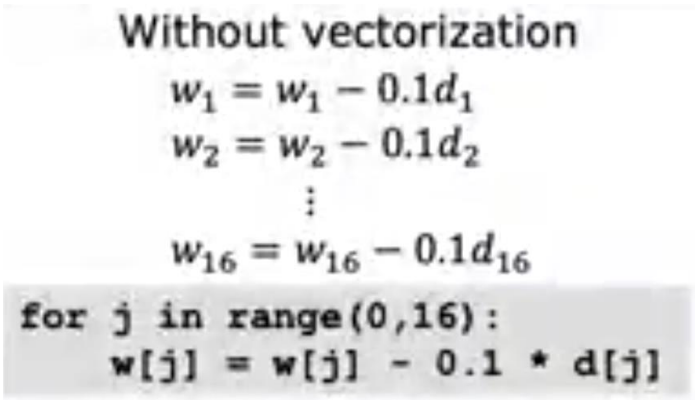


This means that codes with vectorization can perform calculations in much less time than codes without vectorization. This matters more when you're running algorithms on large data sets or trying to train large models, which is often the case with machine learning. That's why being able to vectorize implementations of learning algorithms, has been a key step to getting learning algorithms to run efficiently, and therefore scale well to large datasets that many modern machine learning algorithms now have to operate on.

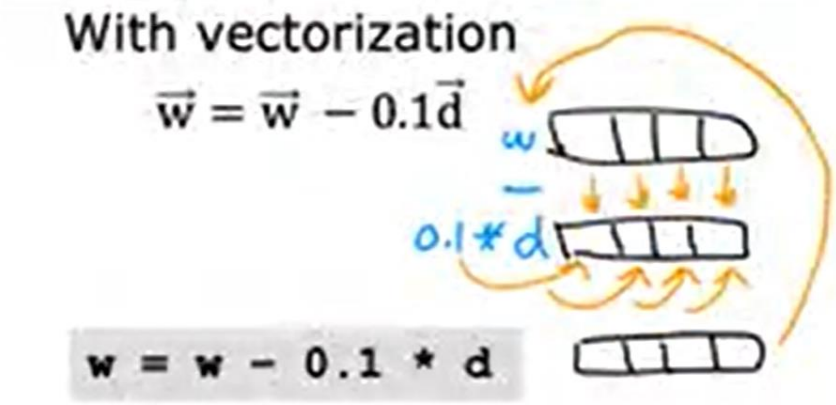


Now, let's take a look at a concrete example of how this helps with implementing multiple linear regression and this linear regression with multiple input features. Say you have a problem with 16 features and 16 parameters, w1 through w16, in addition to the parameter b. You calculate it 16 derivative terms for these 16 weights and codes, maybe you store the values of w and d in two np.arrays, with d storing the values of the derivatives. For this example, I'm just going to ignore the parameter b.

Now, you want to compute an update for each of these 16 parameters. W_j is updated to w_j minus the learning rate, say 0.1, times d_j, for j from 1 through 16. Encodes without vectorization, you would be doing something like this. Update w1 to be w1 minus the learning rate 0.1 times d1, next, update w2 similarly, and so on through w16, updated as w16 minus 0.1 times d16.



Encodes without vectorization, you would be doing something like this. Update w1 to be w1 minus the learning rate 0.1 times d1, next, update w2 similarly, and so on through w16, updated as w16 minus 0.1 times d16. Encodes without vectorization, you can use a for loop like this for j in range 016, that again goes from 0-15, said w_j equals w_j minus 0.1 times d_j.



In contrast, with factorization, you can imagine the computer's parallel processing hardware like this. It takes all 16 values in the vector w and subtracts in parallel, 0.1 times all 16 values in the vector d, and assign all 16 calculations back to w all at the same time and all in one step. In code, you can implement this as follows, w is assigned to w minus 0.1 times d.

Behind the scenes, the computer takes these NumPy arrays, w and d, and uses parallel processing hardware to carry out all 16 computations efficiently. Using a vectorized implementation, you should get a much more efficient implementation of linear regression. Maybe the speed difference won't be huge if you have 16 features, but if you have thousands of features and perhaps very large training sets, this type of vectorized implementation will make a huge difference in the running time of your learning algorithm. It could be the difference between codes finishing in one or two minutes, versus taking many hours to do the same thing.