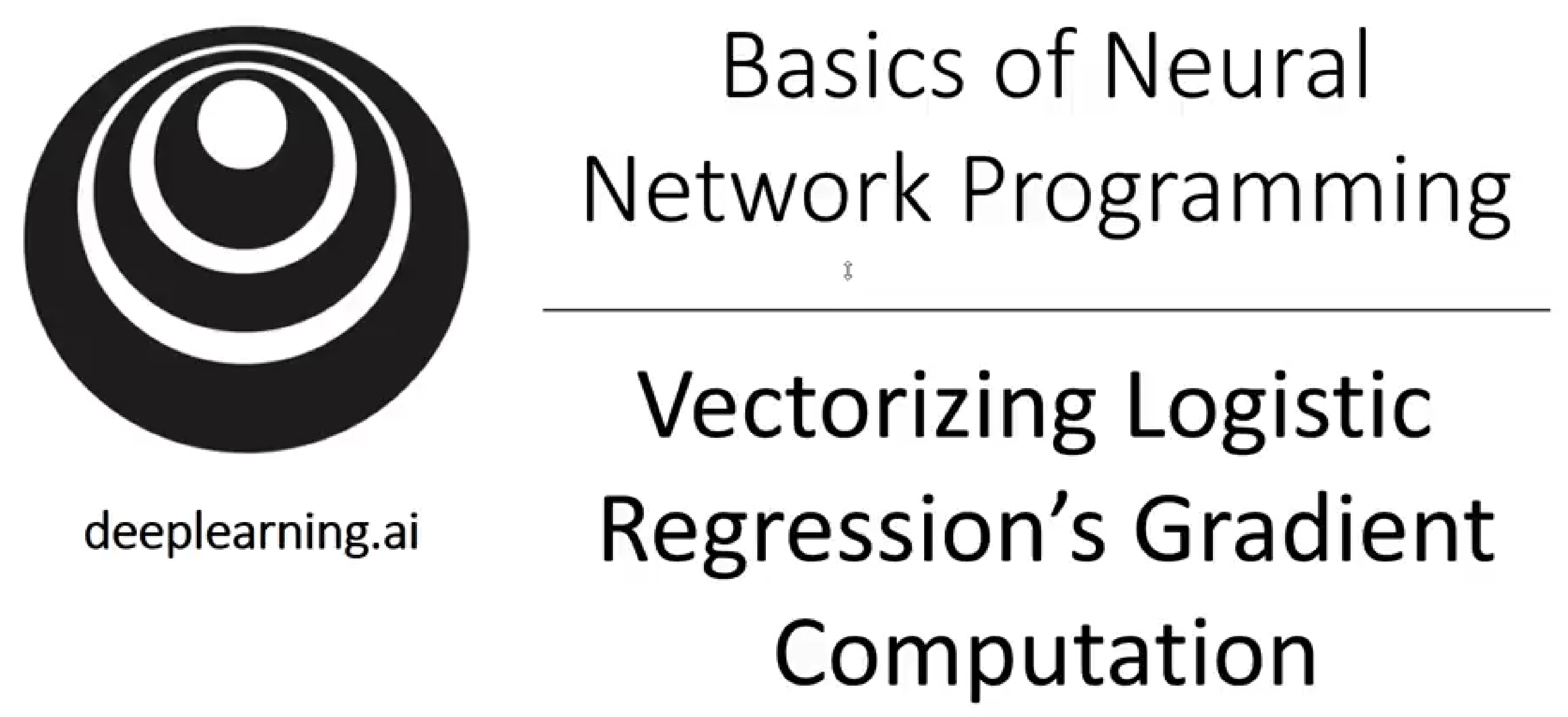

deeplearning.ai

Basics of Neural Network Programming

Vectorizing Logistic Regression's Gradient Computation

Vectorizing Logistic Regression's Gradient Output

Have a question? Discuss this lecture in the week forums.

Interactive Transcript

English

Help us translate!

0:00
In the previous video, you saw how you can use vectorization to compute their predictions. The lowercase a's for an entire training set O at the same time. In this video, you see how you can use vectorization to also perform the gradient computations for all M training samples. Again, all sort of at the same time. And then at the end of this video, we'll put it all together and show how you can derive a very efficient implementation of logistic regression. So, you may remember that for the gradient computation, what we did was we computed dz1 for the first example, which could be a1 minus y1 and then dz2 equals a2 minus y2 and so on. And so on for all M training examples. So, what we're going to do is define a new variable, dZ is going to be dz1, dz2, dzm. Again, all the D lowercase z variables stacked horizontally. So, this would be 1 by m matrix or alternatively a m dimensional row vector. Now recall that from the previous slide, we'd already figured out how to compute capital A which was this: a1 through am and we had to find capital Y as y1 through ym. Also you know, stacked horizontally. So, based on these definitions, maybe you can see for yourself that dz can be computed as just A minus Y because it's going to be equal to a1 - y1. So, the first element, a2 - y2, so in the second element and so on. And, so this first element a1 - y1 is exactly the definition of dz1. The second element is exactly the definition of dz2 and so on. So, with just one line of code, you can compute all of this at the same time. Now, in the previous implementation, we've gotten rid of one full loop already but we still had this second full loop over 20 examples. So we initialize dw to zero to a vector of zeroes. But then we still have to loop over 20 examples where we have dw plus equals x1 times dz1, for the first training example dw plus equals x2 dz2 and so on. So we do the M times and then dw divide equals by M and similarly for B, right? db was initialized as 0 and db plus equals dz1. db plus equals dz2 down to you know dz(m) and db divide equals M. So that's what we had in the previous implementation. We'd already got rid of one full loop. So, at least now dw is a vector and we went separately updating dw1, dw2 and so on. So, we got rid of that already but we still had the full loop over the M examples in the training set. So, let's take these operations and vectorize them. Here's what we can do, for the vectorize implementation of db was doing is

basically summing up, all of these dzs and then dividing by m. So, db is basically one over a m, sum from I equals once through m of dzi and well all the dzs are in that row vector and so in Python, what you do is implement you know, 1 over a m times np. sum of the z. So, you just take this variable and call the np. sum function on it and that would give you db. How about dw or just write out the correct equations who can verify is the right thing to do. DW turns out to be one over M, times the matrix X times dz transpose. And, so kind of see why that's the case. This equal to one of m then the matrix X's, x1 through xm stacked up in columns like that and dz transpose is going to be dz1 down to dz(m) like so. And so, if you figure out what this matrix times this vector works out to be, it is turns out to be one over m times x1 dz1 plus... plus xm dzm. And so, this is a n/1 vector and this is what you actually end up with, with dw because dw was taking these you know, xi dzi and adding them up and so that's what exactly this matrix vector multiplication is doing and so again, with one line of code you can compute dw. So, the vectorized implementation of the derivative calculations is just this, you use this line to implement db and use this line to implement dw and notice that with all the full loop over the training set, you can now compute the updates you want to your parameters. So now, let's put all together into how you would actually implement logistic regression. So, this is our original, highly inefficient non vectorize implementation. So, the first thing we've done in the previous video was get rid of this volume, right? So, instead of looping over dw1, dw2 and so on, we have replaced this with a vector value dw which is dw+= xi, which is now a vector times dz(i). But now, we will see that we can also get rid of not just full loop of row but also get rid of this full loop. So, here is how you do it. So, using what we have from the previous slides, you would say, capitalZ, Z equal to w transpose X + B and the code you is write capital Z equals np. w transpose X + B and then a equals sigmoid of capital Z. So, you have now computed all of this and all of this for all the values of I. Next on the previous slide, we said you would compute the z equals A - Y. So, now you computed all of this for all the values of i. Then, finally dw equals 1/m x dz transpose and db equals 1/m of you know, np. sum dz. So, you've just done front propagation and back propagation, really computing the predictions and computing the derivatives on all M training examples without using a full loop. And so the gradient descent update then would be you know W gets updated as w minus the learning rate times dw which was just computed above and B is update as B minus the learning rate times db. Sometimes it's pretty close to notice that it is an assignment, but I guess I haven't been totally consistent with that notation. But with this, you have just implemented a single elevation of gradient descent for logistic regression. Now, I know I said that we should get rid of explicit full loops whenever you can but if you want to implement multiple adjuration as a gradient descent then you still need a full loop over the number of iterations. So, if you want to have a thousand deliberations of gradient descent, you might still need a full loop over the iteration number. There is an outermost full loop like that then I don't think there is any way to get rid of that full loop. But I do think it's incredibly cool that you can implement at least one iteration of gradient descent without needing to use a full loop. So, that's it you now have a highly vectorize and highly efficient implementation of gradient descent for logistic regression. There is just one more detail that I want to talk about in the next video, which is in our description here I briefly alluded to this technique called broadcasting. Broadcasting turns out to be a technique that Python and numpy allows you to use to make certain parts of your code also much more efficient. So, let's see some more details of broadcasting in the next video.

Downloads

Lecture Video mp4	
Subtitles (English) WebVTT	
Transcript (English) txt	

Would you like to help us translate the transcript and subtitles into additional languages?