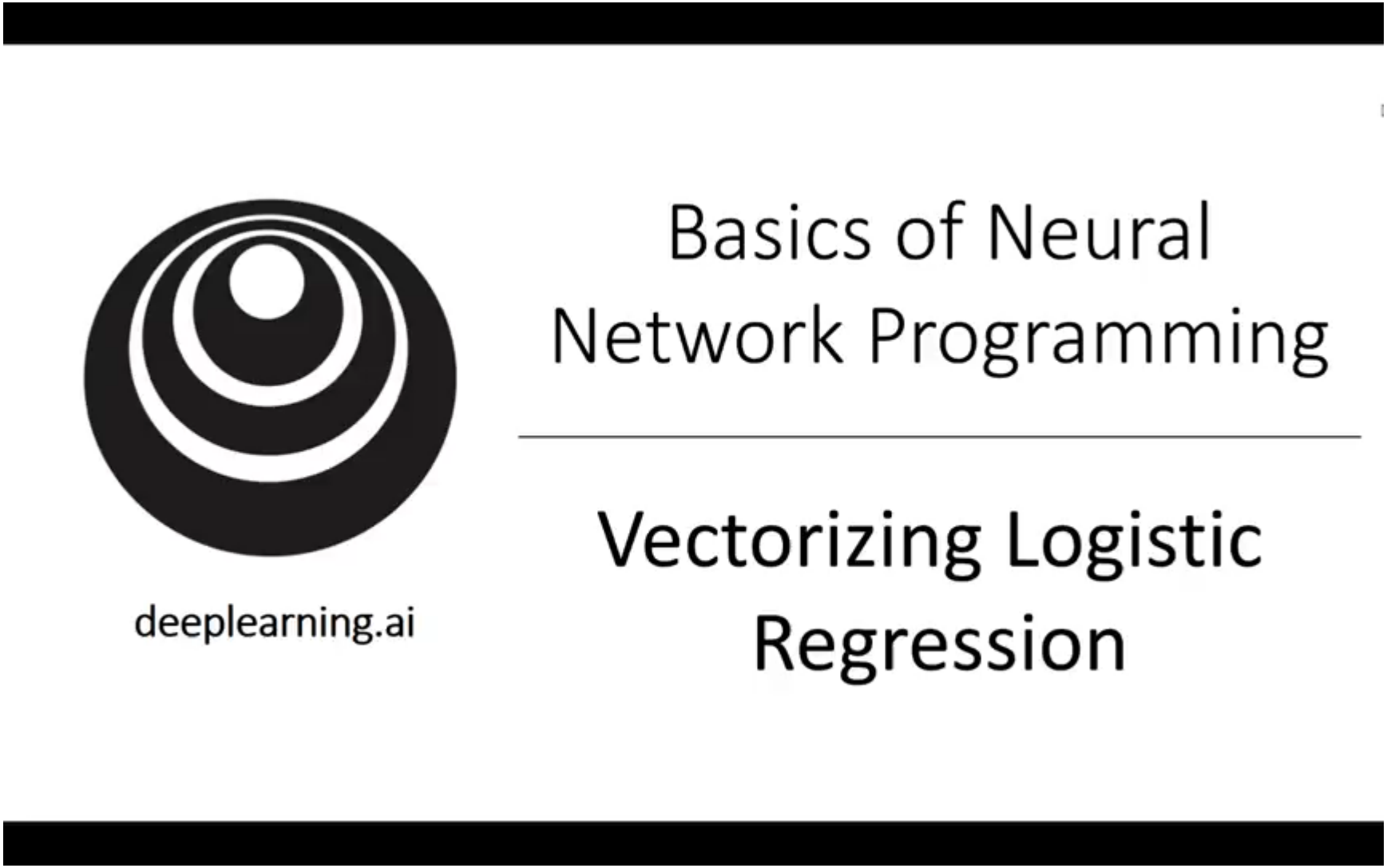

Vectorizing Logistic Regression

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

Interactive Transcript

English

Help us translate!

0:00
We have talked about how vectorization lets you speed up your code significantly. In this video, we'll talk about how you can vectorize the implementation of logistic regression, so they can process an entire training set, that is implement a single elevation of grading descent with respect to an entire training set without using even a single explicit for loop. I'm super excited about this technique, and when we talk about neural networks later without using even a single explicit for loop. Let's get started. Let's first examine the four propagation steps of logistic regression. So, if you have M training examples, then to make a prediction on the first example, you need to compute that, compute Z. I'm using this familiar formula, then compute the activations, you compute [inaudible] in the first example. Then to make a prediction on the second training example, you need to compute that. Then, to make a prediction on the third example, you need to compute that, and so on. And you might need to do this M times, if you have M training examples. So, it turns out, that in order to carry out the four propagation step, that is to compute these predictions on our M training examples, there is a way to do so, without needing an explicit for loop. Let's see how you can do it. First, remember that we defined a matrix capital X to be your training inputs, stacked together in different columns like this. So, this is a matrix, that is a NX by M matrix. So, I'm writing this as a Python draw pie shape, this just means that X is a NX by M dimensional matrix. Now, the first thing I want to do is show how you can compute Z1, Z2, Z3 and so on, all in one step, in fact, with one line of code. So, I'm going to construct a 1 by M matrix that's really a row vector while I'm going to compute Z1, Z2, and so on, down to ZM, all at the same time. It turns out that this can be expressed as W transpose to capital matrix X plus and then this vector B, B and so on. B, where this thing, this B, B, B, B, B thing is a 1xM vector or 1xM matrix or that is as a M dimensional row vector. So hopefully there you are with matrix multiplication. You might see that W transpose X1, X2 and so on to XM, that W transpose can be a row vector. So this W transpose will be a row vector like that. And so this first term will evaluate to W transpose X1, W transpose X2 and so on, dot, dot, dot, W transpose XM, and then we add this second term B, B, B, and so on, you end up adding B to each element. So you end up with another

1xM vector. Well that's the first element, that's the second element and so on, and that's the nth element. And if you refer to the definitions above, this first element is exactly the definition of Z1. The second element is exactly the definition of Z2 and so on. So just as X was once obtained, when you took your training examples and stacked them next to each other, stacked them horizontally. I'm going to define capital Z to be this where you take the lowercase Z's and stack them horizontally. So when you stack the lower case X's corresponding to a different training examples, horizontally you get this variable capital X and the same way when you take these lowercase Z variables, and stack them horizontally, you get this variable capital Z. And it turns out, that in order to implement this, the non-pie command is capital Z equals NP dot W dot T, that's W transpose X and then plus B. Now there is a subtlety in Python, which is at here B is a real number or if you want to say you know 1x1 matrix, is just a normal real number. But, when you add this vector to this real number, Python automatically takes this real number B and expands it out to this 1XM row vector. So in case this operation seems a little bit mysterious, this is called broadcasting in Python, and you don't have to worry about it for now, we'll talk about it some more in the next video. But the takeaway is that with just one line of code, with this line of code, you can calculate capital Z and capital Z is going to be a 1XM matrix that contains all of the lower cases Z's. Lowercase Z1 through lower case ZM. So that was Z, how about these values A. What we like to do next, is find a way to compute A1, A2 and so on to AM, all at the same time, and just as stacking lowercase X's resulted in capital X and stacking horizontally lowercase Z's resulted in capital Z, stacking lower case A, is going to result in a new variable, which we are going to define as capital A. And in the program assignment, you see how to implement a vector valued sigmoid function, so that the sigmoid function, inputs this capital Z as a variable and very efficiently outputs capital A. So you see the details of that in the programming assignment. So just to recap, what we've seen on this slide is that instead of needing to loop over M training examples to compute lowercase Z and lowercase A, one of the time, you can implement this one line of code, to compute all these Z's at the same time. And then, this one line of code, with appropriate implementation of lowercase Sigma to compute all the lowercase A's all at the same time. So this is how you implement a vectorize implementation of the four propagation for all M training examples at the same time. So to summarize, you've just seen how you can use vectorization to very efficiently compute all of the activations, all the lowercase A's at the same time. Next, it turns out, you can also use vectorization very efficiently to compute the backward propagation, to compute the gradients. Let's see how you can do that, in the next video.

Downloads

Subtitles (English)	WebVTT
Transcript (English)	txt
Lecture Slides	pptx

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