More Vectorization Examples

(DESCRIPTION)
Text, deep learning dot AI. Basics of neural network programming. More vectorization examples

(SPEECH)
In the previous video you saw a few examples of how vectorization, by using built in functions and by avoiding explicit for loops, allows you to speed up your code significantly.

Let's look at a few more examples.

(DESCRIPTION)
Text, neural network programming guideline

(SPEECH)
The rule of thumb to keep in mind is, when you're programming your new networks, or when you're programming just a regression, whenever possible avoid explicit for-loops.

And it's not always possible to never use a for-loop, but when you can use a built in function or find some other way to compute whatever you need, you'll often go faster than if you have an explicit for-loop.

Let's look at another example.

If ever you want to compute a vector u as the product of the matrix A, and another vector v, then the definition of our matrix multiply is that your Ui is equal to sum over j,, Aij, Vj.

That's how you define Ui.

And so the non-vectorized implementation of this would be to set u equals NP.zeros, it would be n by 1.

For i, and so on.

For j, and so on..

And then u[i] plus equals a[i][j] times v[j].

So now, this is two for-loops, looping over both i and j.

(DESCRIPTION)
Second for loop is nested inside the first for loop

(SPEECH)
So, that's a non-vectorized version, the vectorized implementation which is to say u equals np dot (A,v).

And the implementation on the right, the vectorized version, now eliminates two different for-loops, and it's going to be way faster.

Let's go through one more example.

Let's

(DESCRIPTION)
Text, vectors and matrix valued functions

(SPEECH)
say you already have a vector, v, in memory and you want to apply the exponential operation on every element of this vector v.

So you can put u equals the vector, that's e to the v1, e to the v2, and so on, down to e to the vn.

So

(DESCRIPTION)
Code that initializes u to zeros, then explicitly loops over it to calculate each element based on the corresponding element of v

(SPEECH)
this would be a non-vectorized implementation, which is at first you initialize u to the vector of zeros.

And then you have a for-loop that computes the elements one at a time.

But it turns out that Python and NumPy have many built-in functions that allow you to compute these vectors with just a single call to a single function.

So what I would do to implement this is import numpy as np, and then what you just call u = np.exp(v).

And so, notice that, whereas previously you had that explicit for-loop, with just one line of code here, just v as an input vector u as an output vector, you've gotten rid of the explicit for-loop, and the implementation on the right will be much faster that the one needing an explicit for-loop.

In fact, the NumPy library has many of the vector value functions.

So np.log (v) will compute the element-wise log, np.abs computes the absolute value, np.maximum computes the element-wise maximum to take the max of every element of v with 0.

v**2 just takes the element-wise square of each element of v.

One over v takes the element-wise inverse, and so on.

So, whenever you are tempted to write a for-loop take a look, and see if there's a way to call a NumPy built-in function to do it without that for-loop.

So,

(DESCRIPTION)
Text, logistic regression derivatives

(SPEECH)
let's take all of these learnings and apply it to our logisti regression gradient descent implementation, and see if we can at least get rid of one of the two for-loops we had.

So

(DESCRIPTION)
Num Py code that initializes all its variables to zero, then explicitly for loops over . Within the body of the loop, operations are written out repeatedly for each of the features of a vector, constituting an implicit inner loop

(SPEECH)
here's our code for computing the derivatives for logistic regression, and we had two for-loops.

One was this one up here, and the second one was this one.

So in our example we had nx equals 2, but if you had more features than just 2 features then you'd need have a for-loop over dw1, dw2, dw3, and so on.

So its as if there's actually a 4j equals 1, 2, and x.

dWj gets updated.

So we'd like to eliminate this second for-loop.

That's what we'll do on this slide.

So the way we'll do so is that instead of explicitly initializing dw1, dw2, and so on to zeros, we're going to get rid of this and instead make dw a vector.

(DESCRIPTION)
Crossing out variable initializations at top

(SPEECH)
So we're going to set dw equals np.zeros, and let's make this a nx by 1, dimensional vector.

Then, here, instead of this for loop over the individual components, we'll just use this vector value operation, dw plus equals xi times dz(i).

And

(DESCRIPTION)
Crossing out inner implicit loop

(SPEECH)
then finally, instead of this, we will just have dw divides equals m.

So now we've gone from having two for-loops to just one for-loop.

We still have this one for-loop that loops over the individual training examples.

So I hope this video gave you a sense of vectorization.

And by getting rid of one for-loop your code will already run faster.

But it turns out we could do even better.

So the next video will talk about how to vectorize logistic aggression even further.

And you see a pretty surprising result, that without using any for-loops, without needing a for-loop over the training examples, you could write code to process the entire training sets.

So, pretty much all at the same time.

So, let's see that in the next video.