Gradient checking

(DESCRIPTION)
Text, deep learning dot AI. Setting up your optimization problem. Gradient checking

(SPEECH)
Gradient checking is a technique that's helped me save tons of time, and helped me find bugs in my implementations of back propagation many times.

Let's see how you could use it too to debug, or to verify that your implementation and back process correct.

So

(DESCRIPTION)
Text, gradient check for a neural network. Take W one, B one, up through W L, B L, and reshape into a big vector theta

(SPEECH)
your new network will have some sort of parameters, W1, B1 and so on up to WL bL.

So to implement gradient checking, the first thing you should do is take all your parameters and reshape them into a giant vector data.

So what you should do is take W which is a matrix, and reshape it into a vector.

You gotta take all of these Ws and reshape them into vectors, and then concatenate all of these things, so that you have a giant vector theta.

Giant

(DESCRIPTION)
Underlining W parameters

(SPEECH)
vector pronounced as theta.

So we say that the cos function J being a function of the Ws and Bs, You would now have the cost function J being just a function of theta.

(DESCRIPTION)
J written two ways, either taking all the parameters W and b for each index, or just taking the comprehensive vector theta

(SPEECH)
Next, with W and B ordered the same way, you can also take dW[1], db[1] and so on, and initiate them into big, giant vector d theta of the same dimension as theta.

So same as before, we shape dW[1] into the matrix, db[1] is already a vector.

We shape dW[L], all of the dW's which are matrices.

Remember, dW1 has the same dimension as W1.

db1 has the same dimension as b1.

So the same sort of reshaping and concatenation operation, you can then reshape all of these derivatives into a giant vector d theta.

Which has the same dimension as theta.

So

(DESCRIPTION)
All the same concatenation steps as on the original parameters

(SPEECH)
the question is, now, is the theta the gradient or the slope of the cos function J?

So here's how you implement gradient checking, and often abbreviate gradient checking to grad check.

So first we remember that J Is now a function of the giant parameter, theta, right?

So expands to j is a function of theta 1, theta 2, theta 3, and so on.

Whatever's the dimension of this giant parameter vector theta.

So to implement grad check, what you're going to do is implements a loop so that for each I, so for each component of theta, let's compute D theta approx i to b.

And

(DESCRIPTION)
Pseudo-code using index variable i

(SPEECH)
let me take a two sided difference.

So I'll take J of theta.

Theta 1, theta 2, up to theta i.

And we're going to nudge theta i to add epsilon to this.

So just increase theta i by epsilon, and keep everything else the same.

And because we're taking a two sided difference, we're going to do the same on the other side with theta i, but now minus epsilon.

And then all of the other elements of theta are left alone.

(DESCRIPTION)
Numerator, the difference of two applications of J, J with parameters theta one, theta two, et cetera, but theta i plus epsilon. Minus, J with parameters theta one, theta two, et cetera, but theta i minus epsilon

(SPEECH)
And then we'll take this, and we'll divide it by 2 theta.

And

(DESCRIPTION)
Actually divided by the denominator 2 epsilon

(SPEECH)
what we saw from the previous video is that this should be approximately equal to d theta i.

Of which is supposed to be the partial derivative of J or of respect to, I guess theta i, if d theta i is the derivative of the cost function J.

So

(DESCRIPTION)
Partial derivative notation. Delta J over delta theta i

(SPEECH)
what you going to do is you're going to compute to this for every value of i.

And at the end, you now end up with two vectors.

You end up with this d theta approx, and this is going to be the same dimension as d theta.

And both of these are in turn the same dimension as theta.

And what you want to do is check if these vectors are approximately equal to each other.

So, in detail, well how you do you define whether or not two vectors are really reasonably close to each other?

What I do is the following.

I would compute the distance between these two vectors, d theta approx minus d theta, so just the o2 norm of this.

Notice

(DESCRIPTION)
Numerator, magnitude of the difference between the two vectors

(SPEECH)
there's no square on top, so this is the sum of squares of elements of the differences, and then you take a square root, as you get the Euclidean distance.

And

(DESCRIPTION)
Denominator, sum of the magnitudes of the two vectors

(SPEECH)
then just to normalize by the lengths of these vectors, divide by d theta approx plus d theta.

Just take the Euclidean lengths of these vectors.

And the row for the denominator is just in case any of these vectors are really small or really large, your the denominator turns this formula into a ratio.

So we implement this in practice, I use epsilon equals maybe 10 to the minus 7, so minus 7.

And with this range of epsilon, if you find that this formula gives you a value like 10 to the minus 7 or smaller, then that's great.

It means that your derivative approximation is very likely correct.

This is just a very small value.

If it's maybe on the range of 10 to the -5, I would take a careful look.

Maybe this is okay.

But I might double-check the components of this vector, and make sure that none of the components are too large.

And if some of the components of this difference are very large, then maybe you have a bug somewhere.

And if this formula on the left is on the other is -3, then I would wherever you have would be much more concerned that maybe there's a bug somewhere.

But you should really be getting values much smaller then 10 minus 3.

If any bigger than 10 to minus 3, then I would be quite concerned.

I would be seriously worried that there might be a bug.

And I would then, you should then look at the individual components of data to see if there's a specific value of i for which d theta across i is very different from d theta i.

And use that to try to track down whether or not some of your derivative computations might be incorrect.

And after some amounts of debugging, it finally, it ends up being this kind of very small value, then you probably have a correct implementation.

So when implementing a neural network, what often happens is I'll implement foreprop, implement backprop.

And then I might find that this grad check has a relatively big value.

And then I will suspect that there must be a bug, go in debug, debug, debug.

And after debugging for a while, If I find that it passes grad check with a small value, then you can be much more confident that it's then correct.

So you now know how gradient checking works.

This has helped me find lots of bugs in my implementations of neural nets, and I hope it'll help you too.

In the next video, I want to share with you some tips or some notes on how to actually implement gradient checking.

Let's go onto the next video.