Random Initialization

(SPEECH)
When

(DESCRIPTION)
Text, One hidden layer. Neural Network. Random Initialization. Website, deep learning, dot, A.I.

(SPEECH)
you change your neural network, it's important to initialize the weights randomly.

For logistic regression, it was okay to initialize the weights to zero.

But for a neural network of initialize the weights to parameters to all zero and then applied gradient descent, it won't work.

Let's see why.

So

(DESCRIPTION)
New slide, What happens if you initialize weights to zero?

(SPEECH)
you have here two input features, so n0=2, and two hidden units, so n1=2.

And so the matrix associated with the hidden layer, w 1, is going to be two-by-two.

Let's say that you initialize it to all 0s, so 0 0 0 0, two-by-two matrix.

And let's say B1 is also equal to 0 0.

It turns out initializing the bias terms b to 0 is actually okay, but initializing w to all 0s is a problem.

So the problem with this formalization is that for any example you give it, you'll have that a1,1 and a1,2, will be equal, right?

So this activation and this activation will be the same, because both of these hidden units are computing exactly the same function.

And then, when you compute backpropagation, it turns out that dz11 and dz12 will also be the same colored by symmetry, right?

Both of these hidden units will initialize the same way.

Technically, for what I'm saying, I'm assuming that the outgoing weights or also identical.

So that's w2 is equal to 0 0.

But if you initialize the neural network this way, then this hidden unit and this hidden unit are completely identical.

Sometimes you say they're completely symmetric, which just means that they're completing exactly the same function.

And by kind of a proof by induction, it turns out that after every single iteration of training your two hidden units are still computing exactly the same function.

Since [INAUDIBLE] show that dw will be a matrix that looks like this.

Where every row takes on the same value.

So we perform a weight update.

So when you perform a weight update, w1 gets updated as w1- alpha times dw.

You find that w1, after every iteration, will have the first row equal to the second row.

So it's possible to construct a proof by induction that if you initialize all the ways, all the values of w to 0, then because both hidden units start off computing the same function.

And both hidden the units have the same influence on the output unit, then after one iteration, that same statement is still true, the two hidden units are still symmetric.

And therefore, by induction, after two iterations, three iterations and so on, no matter how long you train your neural network, both hidden units are still computing exactly the same function.

And so in this case, there's really no point to having more than one hidden unit.

Because they are all computing the same thing.

And of course, for larger neural networks, let's say of three features and maybe a very large number of hidden units, a similar argument works to show that with a neural network like this.

[INAUDIBLE] drawing all the edges, if you initialize the weights to zero, then all of your hidden units are symmetric.

And no matter how long you're upgrading the center, all continue to compute exactly the same function.

So that's not helpful, because you want the different hidden units to compute different functions.

The solution

(DESCRIPTION)
New slide, Random initialization.

(SPEECH)
to this is to initialize your parameters randomly.

So here's what you do.

You can set w1 = np.random.randn.

This generates a gaussian random variable (2,2).

And then usually, you multiply this by very small number, such as 0.01.

So you initialize it to very small random values.

And then b, it turns out that b does not have the symmetry problem, what's called the symmetry breaking problem.

So it's okay to initialize b to just zeros.

Because so long as w is initialized randomly, you start off with the different hidden units computing different things.

And so you no longer have this symmetry breaking problem.

And then similarly, for w2, you're going to initialize that randomly.

And b2, you can initialize that to 0.

So you might be wondering, where did this constant come from and why is it 0.01?

Why not put the number 100 or 1000?

Turns out that we usually prefer to initialize the weights to very small random values.

Because if you are using a tanh or sigmoid activation function, or the other sigmoid, even just at the output layer.

If the weights are too large, then when you compute the activation values, remember that z[1]=w1 x + b.

And then a1 is the activation function applied to z1.

So if w is very big, z will be very, or at least some values of z will be either very large or very small.

And so in that case, you're more likely to end up at these fat parts of the tanh function or the sigmoid function, where the slope or the gradient is very small.

Meaning that gradient descent will be very slow.

So learning was very slow.

So just a recap, if w is too large, you're more likely to end up even at the very start of training, with very large values of z.

Which causes your tanh or your sigmoid activation function to be saturated, thus slowing down learning.

If you don't have any sigmoid or tanh activation functions throughout your neural network, this is less of an issue.

But if you're doing binary classification, and your output unit is a sigmoid function, then you just don't want the initial parameters to be too large.

So that's why multiplying by 0.01 would be something reasonable to try, or any other small number.

And same for w2, right?

This can be random.random.

I guess this would be 1 by 2 in this example, times 0.01.

Missing an s there.

So finally, it turns out that sometimes they can be better constants than 0.01.

When you're training a neural network with just one hidden layer, it is a relatively shallow neural network, without too many hidden layers.

Set it to 0.01 will probably work okay.

But when you're training a very very deep neural network, then you might want to pick a different constant than 0.01.

And in next week's material, we'll talk a little bit about how and when you might want to choose a different constant than 0.01.

But either way, it will usually end up being a relatively small number.

So that's it for this week's videos.

You now know how to set up a neural network of a hidden layer, initialize the parameters, make predictions using.

As well as compute derivatives and implement gradient descent, using backprop.

So that, you should be able to do the quizzes, as well as this week's programming exercises.

Best of luck with that.

I hope you have fun with the problem exercise, and look forward to seeing you in the week four materials.