Effects of randomizing the order of inputs to a neural network

For my Advanced Algorithms and Data Structures class, my professor asked us to pick any topic that interested us. He also told us to research it and to try and implement a solution in it. I chose Neural Networks because it's something that I've wanted to learn for a long time.

I've been able to implement an AND, OR, and XOR using a neural network whose neurons use a step function for the activator. After that I tried to implement a back-propagating neural network that learns to recognize the XOR operator (using a sigmoid function as the activator). I was able to get this to work 90% of the time by using a 3-3-1 network (1 bias at the input and hidden layer, with weights initialized randomly). At other times it seems to get stuck in what I think is a local minima, but I am not sure (I've asked questions on this before and people have told me that there shouldn't be a local minima).

The 90% of the time it was working, I was consistently presenting my inputs in this order: `[0, 0], [0, 1], [1, 0], [1, 0]` with the expected output set to `[0, 1, 1, 0]`. When I present the values in the same order consistently, the network eventually learns the pattern. It actually doesn't matter in what order I send it in, as long as it is the exact same order for each epoch.

I then implemented a randomization of the training set, and so this time the order of inputs is sufficiently randomized. I've noticed now that my neural network gets stuck and the errors are decreasing, but at a very small rate (which is getting smaller at each epoch). After a while, the errors start oscillating around a value (so the error stops decreasing).

I'm a novice at this topic and everything I know so far is self-taught (reading tutorials, papers, etc.). Why does the order of presentation of inputs change the behavior of my network? Is it because the change in error is consistent from one input to the next (because the ordering is consistent), which makes it easy for the network to learn?

What can I do to fix this? I'm going over my backpropagation algorithm to make sure I've implemented it right; currently it is implemented with a learning rate and a momentum. I'm considering looking at other enhancements like an adaptive learning-rate. However, the XOR network is often portrayed as a very simple network and so I'm thinking that I shouldn't need to use a sophisticated backpropagation algorithm.

artificial-intelligence | machine-learning | neural-network | xor | backpropagation

asked Nov 12 '11 at 1:49

Vivin Paliath
39.7k 14 89 154

add comment

1 Answer

the order in which you present the observations (input vectors) comprising your training set to the network only matters in one respect--*randomized arrangement* of the observations *according to the response variable* is strongly preferred versus ordered arrangement.

For instance, suppose you have 150 observations comprising your training set, and for each the response variable is one of three class labels (class I, II, or III), such that observations 1-50 are in class I, 51-100 in class II, and 101-50 in class III. What you do not want to do is present them to the network in that order. In other words, you do not want the network to see all 50 observations in class I, then all 50 in class II, then all 50 in class III.

What happened during training your classifier? Well initially you were presenting the four observations to your network, unordered [0, 1, 1, 0].

I wonder what was the ordering of the input vectors in those instances in which your network failed to converge? If it was [1, 1, 0, 0], or [0, 1, 1, 1], this is consistent with this well-documented empirical rule i mentioned above.

On the other hand, i have to wonder whether this rule even applies in your case. The reason is that you have so few training instances that even if the order is [1, 1, 0, 0], training over multiple epochs (which i am sure you must be doing) will mean that this ordering looks more "randomized" rather than the exemplar i mentioned above (i.e., [1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0] is how the network would be presented with the training data over three epochs).

Some suggestions to diagnose the problem:

1. As i mentioned above, look at the ordering of your input vectors in the non-convergence cases--are they sorted by response variable?
2. In the non-convergence cases, look at your weight matrices (i assume you have two of them). Look for any values that are very large (e.g., 100x the others, or 100x the value it was initialized with). Large weights can cause overflow.

answered Nov 12 '11 at 3:32

doug
29.1k 7 70 132

Thanks for the information doug. Again, since I am a novice I'm not very familiar with all the terms. What do you mean by the "non-convergence cases"? As far as the weights are concerned, I have two weight vectors: one for the input to the hidden layer, and one for the input to the output layer. – Vivin Paliath Nov 12 '11 at 18:03

It was a stupid mistake that did me in. The inputs weren't corresponding to the outputs which ensured that the network didn't learn anything! I also modified my backpropagation algorithm to adjust the weights only after the error has been calculated for all nodes in the network. – Vivin Paliath Nov 13 '11 at 1:30

@VivinPaliath "non-convergence" cases just refers to those cases where the network didn't converge on a result--i.e., which data point caused it to get stuck. Let me suggest. Do you know about the neural net comp.ai FAQ? IMO, by far the best NN resource anywhere--most authoritative, most comprehensive, and surprisingly, easiest to read (entirely in FAQ format). faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html. There are 7 parts, you can download all 7 in pdf form. – doug Nov 13 '11 at 1:47

@VivinPaliath glad to know you solve it. So you reordered your input arrays, but forgot to similarly reorder the corresponding targets--i have done that maybe a million times. if you use python or matlab, one easy way to avoid it, just keep inputs + targets together as a single 2D ('data') matrix, so instead of binding each to a separate variable (inputs, targets) just use index/slice notation on the combined data matrix to refer to either, e.g., inputs are data[:,:-1], targets are data[:,-1] – doug Nov 13 '11 at 1:55

Thank you for that link; it looks very helpful! Also, thank you for your suggestions on making my code less error-prone! – Vivin Paliath Nov 13 '11 at 19:00

add comment