CNS 187 - Neural Computation
Problem Sheet 7

Handed out:	10 Nov 2000
Due:	17 Nov 2000

7.1 Interpolation with multilayer networks

In the previous problem set, you wrote a program to implement the backpropagation algorithm to learn the parameters of a multilayer network. The parametrized function corresponding to the two-layer network was more complicated than that for the one-layer network, giving greater approximating ability. The network uses a set of sigmoidal basis functions to approximate the underlying function implied by the (noisy) data. Here we will look at the effect of changing the number of basis functions.

In multi-class classification problems, the desired output is usually binary with only one element positive. However, the backprop algorithm applies equally well to the case where the desired outputs are analog values. In this problem, you will use the code you wrote for the last problem set to interpolate a one-dimension function[1]

1. Run the algorithm on the file `set3` using 2, 8, and 16 hidden units. Try and get the error small but don't waste too much time on this. Remember that the smaller networks may not be able to interpolate the data exactly. Typical stepsize values are between 0.1 to 0.5. Typical momentum values are between 0.95 to 0.8. Typical numbers of iterations are 10000 to 100000. If your code is like our code, the higher values should work, though perhaps they're overkill.
2. Plot out the data and the *interpolant*, the function implemented by the network, i.e. plot your final network's output $y(x)$ versus x. How does the interpolant change for the three cases of varying numbers of hidden units? Report the final value of E at the end of training for each interpolant. Which interpolant seems like the best solution? Why?

Here are a few suggestions for debugging.

- Include an option in your code that will allow you to plot the components of the interpolation function, $V_{1j}\sigma(W_{j1}x)$. This may help you think about parameter ranges, but don't let it distract you, since such an option is not central to the requirements of this problem.

[1] Because of the sigmoid at the output layer, the output of such a function is restricted to [0,1]. Therefore, if you have training data with desired output over a larger range, you must rescale that training data to [0,1]. Another option is to forgo the nonlinearity on the output units... which yields a slightly simpler gradient calculation.

- Randomize the weights (and biases) in the range $[-1, 1]$. Use the suggestion above to see why this is an appropriate range and what can happen if your initial range is too small or to large.
- If E is not going down steadily, stop and choose another initial condition. Be aware, however, that gradient descent can be a slow process, even with momentum. 100,000 iterations is not unreasonable. Typical stepsize values are between 0.01 to 0.5. Typical momentum values are between 0.99 to 0.5 (with larger values initially).

7.2 Generalization

The last set of data was a noisy sample of an unknown underlying function. We are interested learning the underlying function, but there are often too few samples to adequately constrain the solution space. In speech recognition, for example, the *training* data often consist of only a few examples of each word. A speech recognition algorithm must *generalize* from these examples in order to correctly classify words from arbitrary speakers. One way to compare solutions is to compare generalization ability.

1. See how well the solutions obtained in the last problem generalize by calculating E for the data set `set5`. (At this point you should double-check to make sure your calculation of E reports the average error *per* data point.) This set of data is another noisy sample from the same underlying function. Do *not* update any weights before computing E on `set5`. Of the three networks you implemented (2, 8, and 16 hidden units), which network generalized the best?
2. Now combine the two datasets `set3` and `set4` and rerun the backprop algorithm again using networks of 2, 8, and 16 hidden units. For each solution, graph the interpolant and data, and report the value of E at the end of training.
3. Test the generalization of these new solutions on the file `set5`. Which solution generalized the best? Why? Comment on how the choice of architecture corresponds to an *a priori* assumption of structure in the data. Optionally, try to guess the underlying function.
4. If you looked at what the network computes *before* training has achieved its local minimum of E, you might have noticed that the solution sometimes looks *better* than the full-trained network. To make good on this observation, modify `bprop2` so that it takes as arguments not only `X` and `D`, but also `Xv` and `Dv`: validation data. Do the training based entirely on `X` and `D`, but at each step also compute the error of the network on the validation data. When training is done, return the weights `W` and `V` for the time when the *validation* error was lowest. This is the essence of a technique called *early-stopping*.

   Compare the errors on `set5` of the networks learned by

   - early stopping, where the training set is `set3` and the validation set is `set4`,
   - training on sets `set3` and `set4` combined, with no validation,
   - and training on `set3` only.

Hand in a plot of validation and training error vs iteration number for case (1).

Thus concludes this week's problem set. *"What, I coded up a completely general 2 layer backprop algorithm, and I don't try it on anything with more than 2 inputs and 16 hidden units?!"* you say? *"C'mon, give me some real data!"* Well, go at it, if you've got the gusto, even more data can be found on various Web sites. In defense of simple networks, the purpose of this problem set has been to clearly understand the basics of what's going on in backprop – which can be hard to visuallize when working with 1000-dimensional spaces...