CNS 187 - Neural Computation
Problem Sheet 5

Handed out:	27	Oct 2000
Due:	3	Nov 2000

5.1 Single Layer Networks, HKP chapter 5

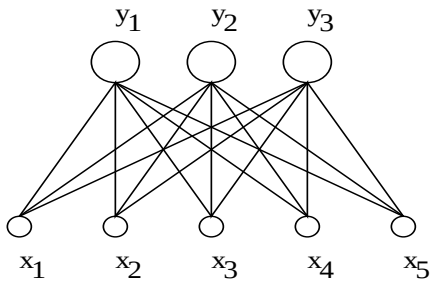

Figure 1: A one layer network.

The network shown in figure 1 is a standard one-layer feed-forward network. The small circles on the bottom are the *input* units. The large circles on the top are the *output* units. The lines connecting the units are the *weights*. The output of unit i to input pattern μ is given by

$$y_i^\mu = \sigma(\sum_{k=0}^{n} w_{ik} x_k^\mu). \tag{1}$$

where w_{ik} is the weight to unit i to from unit k, and $w_{i0} = \theta_i$ is the *bias* of unit i and x_0 is fixed at -1. The function $\sigma(\cdot)$ is called the activation or gain function; we will use the sigmoid[1]

$$\sigma(s) = \frac{1}{1+e^{-s}}. \tag{2}$$

This network represents a function that maps the n dimensional input space to the m dimensional output space. Any particular setting of the weights will correspond to a particular input-output function. In general, we want to set the weights to approximate some unknown target function. We have only a finite set of examples of this function, called the training set.

Suppose we want the desired output of the network to be d_i^μ on the training inputs x_i^μ, and the weights are to be updated according to

$$w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij}. \tag{3}$$

[1]Sigmoid is a name often used to refer to a general class of functions with the properties of being monotonic, differentiable, and bounded.

1. Verify that a learning rule that performs gradient descent on the error measure

$$E = \frac{1}{2} \sum_{\mu=1}^{N} \sum_{i=1}^{m} (d_i^\mu - y_i^\mu)^2 \tag{4}$$

is given by

$$\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}} \tag{5}$$

$$= \eta \sum_{\mu} (d_i^\mu - y_i^\mu) y_i^\mu (1 - y_i^\mu) x_k^\mu \ . \tag{6}$$

(Hint: express $\sigma'(s)$ as a function of $\sigma(s)$.)

You will now write a MATLAB program to implement this learning rule. Look at the files we provided you this week. The first data set you will be using is `set1.dat`, which is an ASCII file where each line contains one sample (two inputs and two outputs). The two outputs represent class membership; class 1 points have output 1 = 1 and output 2 = 0, while class 2 points have output 2 = 1 and output 1 = 0. MATLAB will read the file into a 100×4 matrix `set1` with the command `load set1.dat`. You can view the data points using `boundary(set1(set1(:,3),1:2)',set1(set1(:,4),1:2)',[])`, which will draw a scatter plot of the class 1 points and the class 2 points, with no decision boundary.

Let `X = set1(:,1:2)'` be a 2×100 matrix of input data, where each column represents a separate sample μ. We can make augmented input, with -1 in the first position of each column, by

```
Xp = [ -ones(1,size(X,2)); X ];
```

and let `D = set1(:,3:4)'` similarly be a 2×100 matrix of the desired output. Observe that the 2×100 matrix of network outputs can be written as

```
Y = 1./(1+exp(-W*Xp));
```

where `W` is a 2×3 matrix.

2. Find a MATLAB expression which computes `dEdW`, the 2×3 matrix representing the gradient $\frac{\partial E}{\partial w_{ik}}$. Note that the sum over all samples can be expressed as a matrix multiply – so this is a one-liner.[2]

3. Complete the function `bprop1.m` to do gradient descent on a 1-layer feed-forward neural network. The function should return a vector of the errors $E(i)$, where $i = 1 \ldots$ `iters`.

[2] If you calculate gradients using a `for` loop, your backprop algorithm will be incredibly slow in MATLAB. It would be fine if you were using C or a similar language.

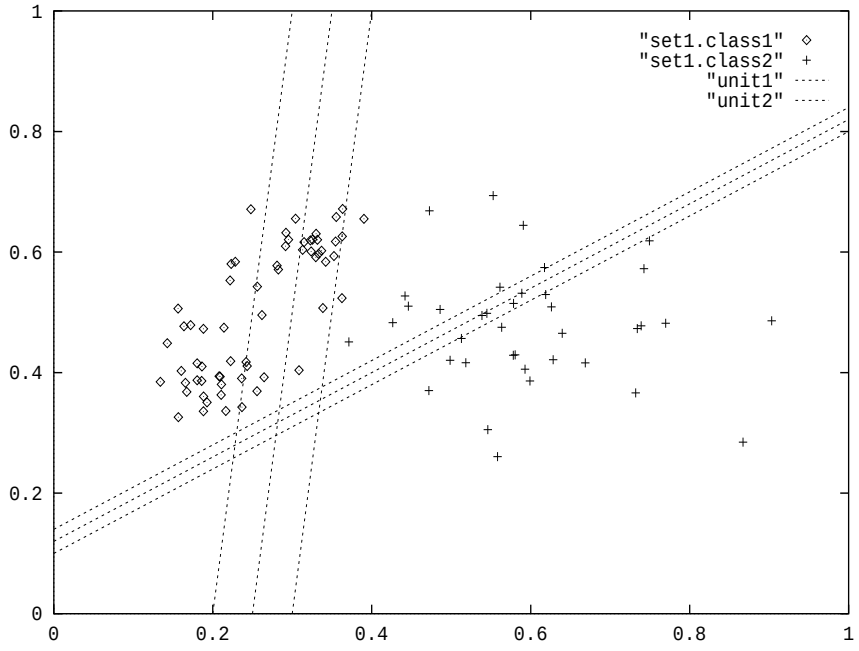

Figure 2: Decision boundaries for a random setting of the weights.

On a successful run, the errors should decrease monotonically. However, you will find that success depends upon the 'learning rate' `eta`, the magnitude of the initial weights, and the number of iterations of weight updates.

Run your program on `set1`. You should be able to see some results for `eta` = .1, initial weights distributed uniformly between [-1,1], and 1000 iterations. The decision boundaries indicate the regions of input space for which the output of the network indicates a particular class. A way of schematically representing the output values for every point in the input space is to overlay a graph of the (2D) input points and, for each output unit, the lines which satisfy the equation $\sigma(s) = 0.2$, $\sigma(s) = 0.5$, and $\sigma(s) = 0.8$. From this, the decision boundaries are easy to see because all you have to do is identify the regions where the output of one unit is > 0.8 and the output of the other unit is < 0.2 and vice versa (remember, the output of the sigmoid is always between zero and one). An example graph is shown in figure 2 which is how the network might divide up the input space after the weights have been initialized. (The function `boundary` is provided to draw these plots.) Be sure to indicate on your decision boundary graph what region of the input space the network would "label" class 1 and what region would be labeled class 2. Use the classification criteria described above, *i.e.* classification is correct if each unit is within 0.2 of it's desired value. Hand in a plot of the data and computed decision boundaries. What percentage of the patterns does the network correctly classify? Describe how the network is performing the classification, *i.e.,* describe the function computed by the learning procedure by graphing the *decision boundaries* of the network.

4. Now run your program on `set2`. What percentage of the patterns does the network

classify correctly? Graph the decision boundaries. Why can't the network learn to classify this set correctly? Would increasing `eta` or `iters` or initializing the weights differently help?

5.2 Multilayer networks, HKP chapter 6, pp 115-123

In the previous problem, you saw that a one layer network represents a limited class of functions. In this problem, you will implement an extension of the learning rule, called *back-propagation*, which minimizes E for multi-layer networks. It can be shown that a two-layer network is sufficient to implement any Boolean function and to approximate any continuous function given a sufficient number of hidden units. Although this may seem like we now have everything we need, we still can't get computers to recognize speech or to classify faces. One possible reason, which we do not address here, is the implicit assumption that a continuous function which solves the problem exists. Another reason, a problem even if the solution *is* continuous, is that the space of functions to search through quickly becomes so enormous that it's impossible to find the tiny subspace of functions that classify the training pattern set correctly *and* do well on patterns that the network hasn't been trained on. The performance of the network on novel patterns is called *generalization* and is something we will consider later.

Here we consider a two-layer network with input units x_i $(i = 1 \ldots n)$, hidden units z_j $(j = 1 \ldots h)$, and output units y_k $(k = 1 \ldots m)$, where

$$z_j = \sigma(r_j), \qquad r_j = \sum_{i=0}^{n} W_{ji} x_i$$

$$y_k = \sigma(s_k), \qquad s_k = \sum_{j=0}^{h} V_{kj} z_j$$

and $x_0 = z_0 = -1$. We use the same sigmoid gain function, as before, for both the hidden units and the output units.

1. Show your derivations for the 4 gradients of E shown below. Then give MATLAB expressions for all 6 expressions shown below.

   - `Z`, the matrix of activity of the hidden units z_j^μ for all data samples,
   - `Y`, the matrix of activity of the output units y_k^μ for all data samples,
   - `eY`, the matrix of deltas $\frac{\partial E}{\partial s_k}$ for all samples,
   - `eZ`, the matrix of deltas $\frac{\partial E}{\partial r_j}$ for all samples,
   - `dEdV`, the matrix of derivatives $\frac{\partial E}{\partial V_{kj}}$,
   - `dEdW`, the matrix of derivatives $\frac{\partial E}{\partial W_{ji}}$.

   Again, each of these expressions can be written as a single line, making use of matrix operations; it helps to write later matrices in terms of previous ones. The matrices `Z`

and `eZ` should be $h \times N$, `Y` and `eY` should be $m \times N$, `dEdV` should be $m \times (h+1)$, and `dEdW` should be $h \times (n+1)$.

2. Use the expressions above to complete the program `bprop2.m` to implement the back-propagation learning algorithm for a 2-layer network, as outlined on page 120 of HKP. The network should have two input units, two hidden units, and two output units; the hidden units and output units should all have biases (don't forget!).

   The gradient descent can be made much more efficient with a little extra effort by adding *momentum* as described on page 123 of HKP. If you don't implement momentum, you could spend a *long* time waiting for the gradient descent procedure to find a minimum; however, you might want to try it without momentum first. If you're curious (optional!) look at other methods in HKP or think of your own.

   Run the program on the same pattern sets used in the last problem, `set1` and `set2`. What percentage of the patterns in each set did the network classify correctly? Why is the two-layer network able to do much better on the second set than the one-layer net?

3. One way to get an idea for how the two-layer network is performing the classification is to use `boundary` to graph the orientation of the sigmoid units in the middle layer over the input set (these indicated the decision boundaries in the one-layer problem). Make this graph for pattern set 2. How is the network performing the classification? Use `boundary` to look at the output units' decision boundaries in terms of the hidden unit activities for class 1 and class 2 input.