COMP30230, Connectionist Computing

March 5th 2013

Due on **April 18th 2013**, by midnight in any time zone of your choice (but start right now, because this will take a significant amount of pondering, and some time coding)
Worth **30%** of the final mark for COMP 30230
One grade off each day late

What to do

Build a Multi-Layer Perceptron MLP, in any programming language of your choice (C, C++, Java, Perl, Python, even Matlab), but without making use of any available neural network/connectionist/machine learning, etc. library.

Your software should be able to:

- Create a new MLP with any given number of inputs, any number of outputs (can be sigmoidal or linear), and any number of hidden units (sigmoidal) in a single layer.
- Initialise the weights of the MLP to small random values
- Predict the outputs corresponding to an input vector
- Implement learning by backpropagation

You need to test your software as follows:

1. Train an MLP with 2 inputs, two hidden units and one output on the following examples (XOR function):
   ((0, 0), 0)
   ((0, 1), 1)
   ((1, 0), 1)
   ((1, 1), 0)

2. At the end of training, check if the MLP predicts correctly all the examples.

3. Generate 50 vectors containing 4 components each. The value of each component should be a random number between -1 and 1. These will be your input vectors. The corresponding output for each vector should be the sin() of the sum of the components. That is, for inputs:
[x1 x2 x3 x4]
the (single component) output should be:
sin(x1+x2+x3+x4)

Now train an MLP with 4 inputs, at least 5 hidden units and one output on 40 of these examples and keep the remaining 10 for testing.

4. What is the error on training at the end? How does it compare with the error on the test set? Do you think you have learned satisfactorily?

Exceptional test (to aim for the max mark)

Train an MLP on the letter recognition set available in the UCI Machine Learning repository:

http://archive.ics.uci.edu/ml/machine-learning-databases/letter-recognition/letter-recognition.data
The first entry of each line is the letter to be recognised (i.e. the target) and the following numbers are attributes extracted from images of the letters (i.e. your input). You can find a description of the set here:
http://archive.ics.uci.edu/ml/datasets/Letter+Recognition

Split the dataset in a training part containing approximately 4/5 of the records, and a testing part containing the rest.
Your MLP should have as many inputs as there are attributes (17), as many hidden units as you want (I suggest ~10) and 26 outputs (one for each letter of the alphabet).
You should train your MLP for at least 1000 epochs. After training, check how well you can classify the data reserved for testing.

What you need to hand in (electronically!)

- Copy of the whole code. I should be able to compile and run this.
- Files containing, for both mandatory tests:
  a. A printout of the error during training.
  b. A brief description of the test you ran, and how you think it worked.
- If you run the exceptional test, send me the same as for the mandatory tests, plus provide the final rate of correct classification for the test set.

Important suggestion!

Take some time to understand exactly what you are supposed to do, and take some time to plan how to do it. Do not rush coding.
You can code the whole assignment in maybe 100 lines of code, and in <2 hours, if you know what you are doing.

How I may do this
I would create an MLP object with the the following attributes and methods (not necessarily an exhaustive list):

Attributes:
NI (number of inputs)
NH (number of hidden units)
NO (number of outputs)
W1[][] (array containing the weights in the lower layer)
W2[][] (array containing the weights in the upper layer)
dW1[][] and dW2[][] (arrays containing the weight *changes* to be applied onto W1 and W2)

Z1[] (array containing the activations for the lower layer – will need to keep track of these for when you have to compute deltas)
Z2[] (array containing the activations for the upper layer – same as above)
H[] (array where the values of the hidden neurons are stored – need these saved to compute dW2)
O[] (array where the outputs are stored)

Methods:
-randomise()
Initialises W1 and W2 to small random values. Also don't forget to set dW1 and dW2 to all zeroes – this could be a good place to do it.

-forward(double* I)
Forward pass. Input vector I is processed to produce an output, which is stored in O[].

-double backwards(double* t)
Backwards pass. Target t is compared with output O, deltas are computed for the upper layer, and are multiplied by the inputs to the layer (the values in H) to produce the weight updates which are stored in dW2 (added to it, as you may want to store these for many examples). Then deltas are produced for the lower layer, and the same process is repeated here, producing weight updates to be added to dW1.
Returns the error on the example.

-updateWeights(double learningRate)
this simply does (component by component, i.e. within for loops):
W1 += learningRate*dW1;
W2 += learningRate*dW2;
dW1 = 0;
dW2 = 0;

Then I would create a program which uses one of these MLP objects (say I call it NN) to train it on the data.
Training would proceed along these lines (pseudo-C code):

```
for (int e=0; e<maxEpochs; e++) {
      error = 0;
      for (int p=0; p< numExamples; p++) {
            NN.forward(example[p].input);
            error += NN.backwards(example[p].output);

            every now and then {
                  updateWeights(some_small_value);
            }
      }
      cout << "Error at epoch " << e << " is " << error << "\n";
}
```

The error that is output should, ideally, get smaller at every epoch. You may have to try different learning rates (too big and training will explode, too small and learning will be very slow), and different "every now and then" – to play safe you can even do the update only once every epoch.