CSCI 4150 Introduction to Artificial Intelligence, Fall 2001
Assignment 7 (80 points): out Monday November 26, due Monday December 3

Late policy deadlines for Problems 1 & 2 are as follows: automatic extension until midnight Monday night December 3; the first tier deadline is midnight Tuesday night December 4; second tier deadline is midnight Thursday night December 6. Corresponding Problem 3 deadlines are the next day at 5pm.

Problems

For this assignment, you will write procedures to train a perceptron using the perceptron learning rule. You would be more likely in practice to be using a multi-layer artifical neural network. However, training a perceptron is much easier, and you should get an idea of how that would work from this assignment. You will also address some of the same issues, such as the learning rate, learning curves, and termination conditions.

Since this is a one week assignment, the amount of code you have to write (and the amount of support code) is fairly small. My solution for Problem 1 is about 14 lines; your solution to Problem 2 might be just as short depending on exactly what approach you take.

1. (30 points) Write the procedure:

   ```
   (perceptron-epoch training-data start-weights alpha)
   ```

   which takes a set of training data (with n numeric attributes), a list of $n + 1$ weights, and a learning rate α, and applies the perceptron learning rule:

   $$\vec{w} \leftarrow \vec{w} + \alpha \times \vec{I} \times err$$

   (where err is the correct output value minus the actual output value) using each example in the training data.

2. (30 points) Write the procedure:

   ```
   (learn-perceptron training-data testing-data)
   ```

   which, given a training data set and a testing data set returns a list of weights for a perceptron. This procedure should train the perceptron on the training data, periodically evaluating its performance on the testing data.

   This problem will be evaluated by measuring the performance of your procedure, taking into account both the accuracy of the resulting perceptron and the amount of computation time required to learn the perceptron. There will be some maximum allowable CPU time within which your procedure must run. (See the assignment web page for details.) You may use any approach for learning a perceptron; however, I will provide some suggestions below.

3. (20 points) Turn in written answers to the following:

   - Describe the approach you used in your `learn-perceptron` procedure, i.e. learning rate value(s), termination condition, initial weights, or any other modifications to the basic algorithm.
   - Using your `learn-perceptron` procedure, measure and plot a learning curve — record data on the performance of the perceptron after each epoch and make a graph of epoch versus performance. (I will provide a data set specifically for this problem.)
   - Explain any deviations of your learning curve from the stereotypical learning curve.

Notes, conventions, and suggestions

- If the training data set has n attributes, your vector of weights should have $n+1$ elements. By convention, the first weight is the threshold level of the perceptron. This implies that a -1 must be prepended to each list of input values before applying the perceptron learning rule. You can see this convention used in the `test-perceptron` procedure in the support code.
- Typically, your initial weights should be small random numbers. I have provided support code to generate a list of random numbers in the range $[-1, 1)$.
- The learning rate should be a small positive number (typically on the order of 0.1 or 0.005). There are two basic approaches to use for the learning rate:
  - use a constant learning rate
  - make the learning rate decrease as you do more epochs
- There is also the question of when to stop learning. There are many approaches to this problem:
  - do a fixed number of epochs
  - learn until the increase in performance falls below some threshold
  - learn until the performance reaches a peak

  (I have provided support code to test a perceptron against a set of testing data.) I'm sure you can think of other approaches... There is no "universal" strategy for this problem.

Support code

The support code for this assignment follows. (Actually, the support code I'll release will check that the inputs to the procedures are valid.)

```
; Compute the output value of a perceptron
;
; The weights and input-vector lists should be the same length.
; Prepending a -1 to the beginning of the inputs to form the input
; vector must be done before this procedure is called.
;
(define (perceptron weights input-vector)
  (if (> (dot-product weights
                      input-vector)
         0)
      1
      -1))

(define (dot-product a b)
  (apply + (map * a b)))

; Accessors for the training/testing data format, assuming it is
; structured as a list of training examples where each example is of
; the form:
;
;    (output (input1 input2 ... inputN))
;
(define td-answer first)
(define td-inputs second)
```

```
;
; Test a perceptron against a set of testing data.
;
; Returns the percentage of examples correctly classified correctly.
;
(define (test-perceptron weights testing-data)
  ; the arguments to the helper procedure are the remaining training
  ; data (td) and the number of examples right and wrong so far...
  (define (tp td right)
    (if (null? td)
        (let ((fraction-correct (/ right (length testing-data))))
          (printif "\nOut of " (length testing-data) " examples, "
                   right " (" (round (* 100 fraction-correct))
                   " percent) were correctly classified.\n")
          (exact->inexact fraction-correct))
        (let* ((correct-answer (td-answer (first td)))
               (input-vector (cons -1 (td-inputs (first td))))
               (answer (perceptron weights input-vector))
               (correct? (= answer correct-answer)))
          (tp (cdr td)
              (if correct?
                  (+ 1 right)
                  right)))))

  (tp testing-data 0))

; produce a list of N random numbers in the range of [-1, 1)
;
(define (n-random#s N)
  (if (zero? N)
      '()
      (cons (- (random 2.0) 1)
            (n-random#s (- N 1)))))

; print (using display) all the arguments to this procedure
; (takes a variable number of arguments)
;
(define (print . stuff)
  (if (not (null? stuff))
      (begin
        (display (car stuff))
        (apply print (cdr stuff)))))

(define print-info #t) ; controls how much information is printed...

(define (printif . stuff)
  (if print-info
      (apply print stuff)))
```