COMPUTER SCIENCE 397 (Spring Term 2005)
Neural Networks & Graphical Models
Prof. Levy

Problem Set 1: Perceptron Learning

Due Friday 22 April

1 Reading Assignment: *PDP* Ch. 1; *AIMA* Ch. 19.1-5

2 Programming Assignment: Building a Perceptron

This assignment serves two purposes: (1) Introducing you to Matlab, the numerical programming / visualization environment that we will use throughout this course (2) Building, training, and testing a perceptron, the first neural network model that we learned about in this class. Instead of going through a separate tutorial, you will learn about Matlab by using it to implement the perceptron. A good way to do this assignment is to keep the PERCEPTRON-LEARNING algorithm in front of you as you implement it one piece at a time.

To begin, let's create a simple mapping (input/output relation) for our perceptron to learn. One such mapping that you may already know about is the boolean OR function. This function outputs true (1) when at least one of its inputs is true (1), and outputs false (0) otherwise:

a1	a2	OR(a)
0	0	0
0	1	1
1	0	1
1	1	1

A main feature of Matlab is that we can represent data like the variable `a` above using matrices.[1] Matrices can be created in the Matlab interpreter, or loaded from `.mat` files. For the simple OR map, we will create the Matrix in the Matlab interpreter. Launch Matlab (you will be shown how to do this in class), and create a matrix `a` by typing:

```
>> a = [0 0; 0 1; 1 0; 1 1]
```

[1]Indeed, the name *Matlab* stands for *Mat*rix *lab*oratory.

Matlab should respond with:

```
a =
     0     0
     0     1
     1     0
     1     1
```

As you can see, the semicolon is used to separate rows of the matrix. Knowing this, you should be able to create a four-row, one-column matrix `b`, which will contain the values of `OR(a)`. Matlab should respond with

```
b =

     0
     1
     1
     1
```

Now that we have our input and target matrices, we can begin to encode the PERCEPTRON-LEARNING algorithm. We need to tell Matlab that we are writing a procedure, which in Matlab is called a *function.* Since we don't want to type PERCEPTRON-LEARNING every time we invoke this function, let's just call it `percep`. Like the PERCEPTRON-LEARNING procedure our `percep` function will take two arguments, a matrix I of inputs and a matrix T of targets, and will return matrix w of weights. For the data you've generated above, this would be accomplished by typing the following into the Matlab interpreter:

```
  >> w = percep(a, b)
```

The header (first line) of your function should read as follows:

```
function w = percep(I, T)
```

Type this line into whatever editor you are using (Matlab's editor, emacs, etc.), and save to a file called `percep.m`.[2]

The first line of PERCEPTRON-LEARNING tells us to create an $(n+1) \times m$ matrix of small random numbers, where n is the number of inputs and m is the number of outputs. Since the number of inputs is just the number of columns in `I`, and the number of outputs is the number of columns in `T`, we can use Matlab's `size` function to compute m and n. This function takes a matrix and a dimension and returns the size of the matrix along that dimension:

```
n = size(I, 2);
m = size(T, 2);
```

In a similar way, we can use `size` to compute the total number of patterns p, which is simply the number of rows in I:

[2]In general, it's a very good idea to make the name of the file agree with the name of the function it contains.

```
p = size(I, 1);
```

Now that we have these values, we can use them to create a random matrix using Matlab's `randn` function, which returns a random matrix with a zero mean and standard deviation of 1:

```
w = randn(n+1,m);
```

Now when we run our program by typing `percep(a, b)`, Matlab show us some random numbers, corresponding to the weights `w`. This happens because, unless we put a semicolon after something, Matlab always show us its value.[3]

The second line of the algorithm says to attach a column of 1's to the end of `I`. To understand how we can do this in Matlab, we need to know how Matlab deals with rows and columns. First, we observe that `a`(i, j) refers to the ith row and jth column of matrix `a`. So if we type `a(1,2)` into the interpreter, Matlab should respond with `0`, since that is the value in the first row, second column of the matrix. Second, we note that the colon character can be used as a special index, to refer to an entire row or column all at once. So typing `a(:,2) = 3` will set the entire second column (all rows) of `a` to the value 3. To attach a column of 1's to our matrix `I`, we could therefore say `I(:,3) = 1`. However, this solution will only work when `I` has two columns. Fortunately, Matlab provides a special index `end`, which always refers to the highest index in a row or column. By adding 1 to this index, we can create a new final row or column in a matrix. Hence the next line of our function should be:

```
I(:,end+1) = 1;
```

Now we have to write some code to loop 1000 (or some other convenient number) of times. A *for* loop in Matlab looks pretty similar to pseudocode, except that there's no *do*, and we need to terminate the loop with an `end` statement:

```
for i = 1:1000

end
```

The first line in this loop should set up an $(n + 1) \times m$ matrix of zeros. Matlab provides a `zeros` function to do this for us:

```
for i = 1:1000
  dw = zeros(n+1, m);
end
```

At this point, finishing the `percep` function should be straightforward. The only additional trick we need to know about is how to compute the transpose of a vector, for the line $\Delta\, w \;\leftarrow\; \Delta\, w + D * I_j^T$. Matlab uses a single-quote symbol for this operation, so this line appears as `dw = dw + D*I(j,:)'` in our Matlab code. You should be able to figure out the rest of the code yourself. Note that it is not necessary to write a `return` statement, because the return value `w` is declared at the top of the function.

[3]So leaving out semicolons is a good way to debug your code, but a bad thing to do when you submit it!

Finally, you should write a test function `ptest`, in a separate file `ptest.m`, to test your weights once you have them. This function should take a weights-matrix `w` and an input `I`, and return a vector `O` containing the output computed by the perceptron. Once you have done this, you can easily try out your learning function on the boolean AND mapping as well (`b = [0; 0; 0; 1]`).

3 Advanced Issues

If you have time left after writing `percep.m` and `ptest.m`, try the following additions to your code. *Material in italics contains "thought questions" whose answers you do not need to submit – but they might appear on a quiz or exam!*

3.1 Error plotting

It is often useful to see how the error (target - output difference) changes over the iterations in the outer loop. Matlab's `plot` command allows you to do this with a minimum of fuss. Rather than plotting the difference D on each iteration of the outer (or inner) loop, you can have your `percep` function return not just the weights, but also the errors on each iteration, which you can then plot all at once. The first change you should make is to the header of the program:

```
function [w, e] = percep(I, T)
```

This new header will tell the Matlab interpreter to return two outputs (`w` and `e`) if the user requests two outputs, and to return just the first (`w`) otherwise. So in addition to getting the weights as before:

```
>> w = percep(a, b)
```

you can also obtain and plot the errors:

```
>> [w,e] = percep(a, b);
>> plot(e)
```

Of course, you must also compute the vector e_i, containing the average error (over the four patterns) at each iteration i. Before the inner (`j`) loop, set e_i to 0. Inside the inner loop, add the absolute value of the current difference D to e_i. (We use absolute value because we're interested in the size of the error, not its positive/negative direction.) After the inner loop, divide e_i by the number of patterns p (not strictly necessary, since we're only interested in how the error changes, not its actual value). Then re-run the `percep` command and generate a plot as described above. *Do you really need 1000 iterations, or does the error converge (go to its minimum value) much faster than that?* Change your code based on your answer to this question.

3.2 Learning eXclusive-OR

In addition to the AND and OR functions, a very useful boolean function is eXclusive-OR (XOR), whose output is 1 when its inputs differ, and is 0 otherwise.[4] Create a new target for XOR and generate and plot the error:

[4]For example, you can erase a binary image by XORing it with itself, which is useful for animation.

```
>> b = [0; 1; 1; 0];
>> [w,e] = percep(a, b);
>> plot(e)
```

Does the error converge? If not, what does the Perceptron Convergence Theorem tell us about the possibility of using a perceptron to compute the XOR function?

3.3 Zero initial weights

The first line of the PERCEPTRON-LEARNING algorithm sets the initial network weights to small random numbers close to zero. Why not just set all the weights to zero instead? To get some insight into this question, initialize the weights using `zeros` instead of `randn`, and see what happens when you try to learn XOR. *Does the error value ever change? If not, think about the constant value on which it stays fixed, and what that value represents.*

3.4 Fixing initial conditions for reproducible results

As we will soon see, neural networks is more of an empirical (experimental) science than are most of the other computer science topics you have studied: it is often difficult or impossible to know how well your algorithm will perform on a given data set, without trying it out on that data set. This unpredictability can also make debugging extremely difficult. For this reason, it is important to write your programs in a way that you others can reproduce your results: given the same input, you or someone else should be able to obtain the same output over and over. With random initial conditions (weights), this means running the same experiment many times, and checking the consistency of your results by statistical methods. Even then, debugging can be a nightmare, because you don't know what values to expect until the very end. For this reason, most random-number generators allow you to generate the same set of random numbers as many times as you want, by "seeding" the generator with a fixed value.[5] In the absence of such a value, the generator provides its own, meaningless seed, by using a quasi-random value like the clock time in seconds.

In Matlab, the same `randn` function that you use to generate your initial weights can be used to set the generator seed. Before your current call to `randn`, insert the line `randn('seed', 0)`. Now, running your original (AND or OR) experiment several times should produce the same error signal, which you can verify by plotting. Changing the seed to another value should give a different error signal.

[5] On the other hand, it is wrong, and scientifically dishonest, to report a result that is only valid for a particular seed or set of seeds!