CPSC 420-502: Project 2, Perceptron and Backpropagation

Yoonsuck Choe
Department of Computer Science
Texas A&M University

1 Overview

You will implement perceptron learning from scratch (see section 3 for details), and train it on AND, OR, and XOR functions. Then, you will take an existing backpropagation code (see section 4), train it and test it under different conditions and report your findings. The same three boolean functions will be used for the backpropagation learning. For further details on perceptrons and backpropagation, see the lecture slides and also Hertz et al. (1991). Specific submission instruction will be given in section 5 and section 6.

2 Language and OS

You may use either C/C++, Java, or Lisp. The resulting code should be able to compile and run on the departmental unix hosts (unix.cs.tamu.edu, compute.cs.tamu.edu, interactive.cs.tamu.edu). You may use a different language with a permission from the instructor. You will be asked to do a demo in front of the TA in that case.

To compile your programs other than Lisp (which you already know), see the following instructions.

- C program file: **perceptron.c**
  to compile: **cc -o perceptron perceptron.c -lm**
  to execute: **./perceptron**
- C++ program file: **perceptron.C**
  to compile: **c++ -o perceptron perceptron.C -lm**
  to execute: **./perceptron**
- Java program file: **perceptron.java**
  to compile: **javac perceptron.java**
  to execute: **java perceptron**

The full paths for the compilers are:

- /usr/local/SUNWspro/bin/cc
- /usr/local/bin/c++
- /usr/local/java/bin/javac
- /usr/local/java/bin/java

3 Perceptron

Perceptron activation is defined as:

$$Output = step\left(\sum_{i=0}^{2} W[i] * INP[i]\right), \qquad (1)$$

where $step(X) = 1$ if $X \geq 0$ and $step(X) = 0$ if $X < 0$ (see figure 1).

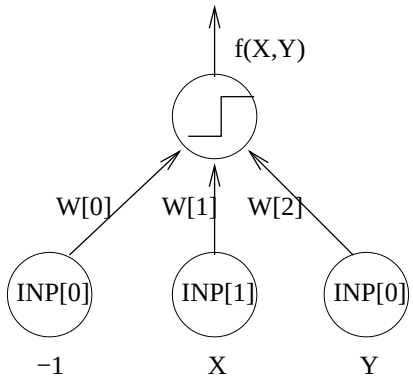

Figure 1: **Perceptron.** The input $INP[0]$ is the *bias unit*, fixed to -1, and has an associated weight $W[0]$, which is the threshold. The two inputs X and Y are given and the output $f(X, Y)$ will be calculating (or attempting to calculate) a boolean function, one of OR, AND, or XOR.

Implement a perceptron with two input units (three, including the bias unit that has a fixed input value -1) and one output unit. Your program should take 5 inputs from the keyboard (i.e. standard input):

1. maximum number of epochs to run (integer),
2. learning rate parameter α value (double),
3. function selection string ("and", "or", and "xor").

For example, a typical run would go like this ($ is the unix prompt):

```
$ ./perceptron
10000 0.0001 and
```

where in the first line is you run your program, and in the second line, you type in the parameters. A pseudocode for perceptron learning is as follows:

1. Initialize weights to random numbers between 0.0 and 1.0.
2. Initialize epoch count to 0.
3. while sum of $(teacher - output)^2$ for all input patterns is not 0 do:
   - for each input-teacher pattern
     - present input and calculate output
     - calculate the $error = teacher - output$
     - update the weights $W[i]$
   - endfor
   - increment epoch count
   - if (epoch count $>$ max epochs), break from while loop
4. Print out the output for the inputs (0,0), (0,1), (1,0), and (1,1).

4 Backpropagation

For backpropagation, you will download the following file (make it one line):

```
http://faculty.cs.tamu.edu/choe/
src/backprop-1.6.tar.gz
```

and run it under different conditions. First, you need to unzip and untar it by running the following:

```
$ tar xzvf backprop-1.6.tar.gz
$ cd backprop
```

then read the `README` file to learn how to compile and run it.

Running the `bp` program (which is generated by compiling) will give you a huge dump on the screen. To selectively view the data you're more interested in, use the `grep` command. For example, to view the progression of error:

```
$ ./bp conf/xor.conf | grep ERR
```

and to view the actual output values for the inputs:

```
$ ./bp conf/xor.conf | grep OUT
```

5 Assignment

This section will detail what you actually have to do and have to submit. All projects should be turned in using the `turnin` command to the folder `420-502`.

5.1 Perceptron

Implement perceptron learning algorithm as detailed in section 3, and with the program conduct the following experiments, and submit the required material, along with the code and the `README` file as usual.

Experiments:

1. Test AND, OR, and XOR for learning rates $\alpha = 0.001$, and 0.0001. For each Boolean function, run the experiment with different initial random weights (use the random number generator function to do this) three times. Discard all runs that ended in 1 epoch (you will see several of these). The total number of trial will thus be 18. Set the max epoch to 10000 for all trials.
2. For each trial, report the following in the README file:
   (a) inital weights,
   (b) after each epoch, save the the connection weights W[0]..W[2] for later use,
   (c) final weights,
   (d) number of epochs taken to complete training if successful (let us call this n), and
   (e) for each epoch, the sum of squared error for all four input patterns.
3. Answer these questions in the README file:
   (a) How dependant is n on the initial weights?
   (b) Is AND more difficult than OR, or vice versa, or are they just the same? why? Run many trials and compare n to get an empirical justification, and then think about why.
   (c) XOR will fail inevitably: what are the output to the four inputs (0,0), (0,1), (1,0), and (1,1) at the end of max epoch? Is there a consistency for differently initialized networks? Run many trials and find out the probability of different outcomes. There are 16 different output combinations for inputs (0,0) (0,1) (1,0) (1,1): 0 0 0 0, 0 0 0 1, 0 0 1 0, ..., 1 1 1 1.
   (d) Do you think perceptron will be able to learn the boolean function $f(X, Y) = \neg(X \wedge Y)$? What do you think is the role of the *sign* of the weights in this case? What is the geometric interpretation?

Table 1: **Boolean Function** $f(X, Y) = \neg(X \wedge Y)$.

X	Y	$\neg(X \wedge Y)$
0	0	1
0	1	0
1	0	0
1	1	0

4. For the longest run for each trial for AND, OR, and XOR, with $\alpha = 0.0001$ generate one graph each (a total of three graphs) containing the class boundary *lines* derived from the connection weights saved from item 2 in the previous list (report these weights in the README file and the line equation derived from it). If you had n epochs, plot 5 lines at an interval of $\frac{n}{5}$, including the initial and the final epochs. For example, if your n was 800, plot for epochs 0, 200, 400, 600, and 800. Save the plot to any image format (jpg, gif, png, etc.) and name them `p-plot0.jpg, p-plot1.jpg,` etc. and submit them separately. See figure 2 for an example plot.

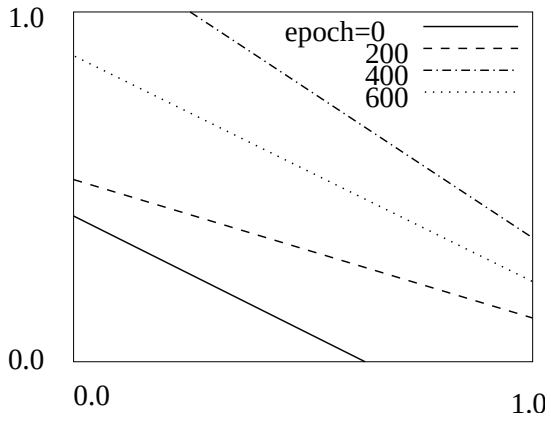

Figure 2: **Example Class Boundary Plot.** The class boundary line derived from the weights are shown after four different epochs: 0, 200, 400, and 600.

5.2 Backpropagation

With the provided code, conduct experiments on AND, OR, and XOR. Note that in the `bp.cc` code, learning rate α is a named `eta`, just in case you want to take a look inside the code.

Experiments:

1. Test AND, OR, and XOR for learning rates $\alpha = 0.01$, 0.001, and 0.0001, and plot the sum of squared error for each trial (a total of 9 trials). A total of 3 plots (for AND, OR, and XOR), with 3 curves each (for three α's) is required. Save the plots in image files and name then `b1-plot0.jpg, b1-plot1.jpg, ....`

2. With learning rate $\alpha = 0.001$, test AND, OR, and XOR, with 1, 2, 3, and 4 hidden units, Plot the sum of squared error for each trial. A total of 3 plots (for AND, OR, and XOR), with 4 curves each (for four different number of hidden units) is required. Save the plots in image files and name then `b2-plot0.jpg, b2-plot1.jpg, ....`

3. For all trials above, count the number of epochs until the end is reached, and measure the time taken using the `timex` unix command:

   ```
   timex ./bp conf/xor.conf
   ```

   Report the number of epochs and time spent for each trial in the README file. How many number of hidden units was the best in your opinion and why?

6 Submission Details

The **due date** is by the date of the final exam: 12/17/2002 8am. There will be absolutely no extensions given the time constraints. Grading criteria will be similar to the previous projects. You must submit the following:

- source code,
- compiled executable binary,
- README file containing material detailed in section 5, and
- plots (image files; include a list of image files and a brief description of each in the README file).

All projects should be turned in using the `turnin` command to the folder `420-502` by 12/17/2002 8am.

References

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation.* Reading, MA: Addison-Wesley.