CS 540: Introduction to Artificial Intelligence

HW #4: Neural Networks, SVMs, and Face Detection

Assigned: March 21
Due: April 12

Late policy:

Homework must be handed in at the start of class on the due date and electronically turned in by this same time.

- If it is 0-24 hours late, including weekend days, a deduction of 10% of the maximum score will be taken off in addition to any points taken off for incorrect answers.
- If it is 24-48 hours late, including weekend days, a deduction of 25% of the maximum score will be taken off in addition to any points taken off for incorrect answers.
- If it is 48-72 hours late, including weekend days, a deduction of 50% of the maximum score will be taken off in addition to any points taken off for incorrect answers.
- If it is more than 72 hours late, you will receive '0' on the assignment.
- In addition, there are 2 'late days' you may use any time throughout the semester. Each late day has to be used as a whole – you can't use only 3 hours of it and "save" 21 hours for later use.

Collaboration policy:

You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TA and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

- Not explicitly tell each other the answer
- Not to copy answers and code fragments from anyone or anywhere
- Not to allow your answers to be copied
- Not to get any code or help off the Internet

In those cases where you work with one or more other people on the discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignments the names of the people you were in discussion with.

Question 1: Representing Boolean Functions using Neural Networks

For each of the following sentences in Propositional Logic, define a Perceptron that correctly computes the desired output, if possible. Let 1 correspond to true and 0 false. Use a simple threshold activation (LTU) function at each unit. Label the weights and the thresholds at each unit in the network. If it is not possible to define the function using a Perceptron, say why not and then give a feedforward network with one hidden layer (using LTUs) that implements it.

[a] $(A \wedge B) \vee (A \wedge \neg B)$
[b] $(A \vee B) \wedge (\neg A \vee \neg B)$
[c] $(A \wedge B) \Leftrightarrow (\neg A \wedge B)$

Question 2: Perceptron Learning Rule

Problem 20.15(a) in the textbook.
Note:

1. Please show me the procedure of calculation of three epoch
2. Assume that alpha = 0.3 and all the perceptrons are initialized to 0.5

Question 3: Neural Networks and Back-Propagation

Consider a learning task where there are two real-valued input features, A and B, and one binary output, C. You decide to use a 2-layer neural network with three hidden units, H1, H2, and H3, defining the hidden layer. The activation function is the sigmoid function defined in Figure 20.16(b) in the textbook. Each input unit is connected to every hidden unit, and each hidden unit is connected to the output unit.

[a] Draw this neural network and initialize all the weights into and out of H1 to 0.4, all of the weights into and out of H2 to -0.1, and all the weights into and out of H3 to 0.2. Initialize the "bias" of each hidden unit and output unit to 0.2. (In practice, these weights would be set to small, randomly chosen, numbers but for grading simplicity we are specifying the initial values.)

[b] Consider the following training example and show how the back-propagation algorithm (Figure 20.25 in the textbook) would change your network's weights and biases. Using a learning rate of $\alpha = 0.2$

$$A = 0.3,\ B = -0.6,\ C = 1$$

Question 4: Support Vector Machines

You are given a data set with a single input feature, X_1, in $\mathbf{R}^1$ and desired classification output, $Y \in \{-1, +1\}$, as shown in Figure 1. The training set contains three examples in class -1, $X_1 = 0$, $X_1 = 3$ and $X_1 = 4$; and two examples in class +1, $X_1 = 1$ and $X_1 = 2$.

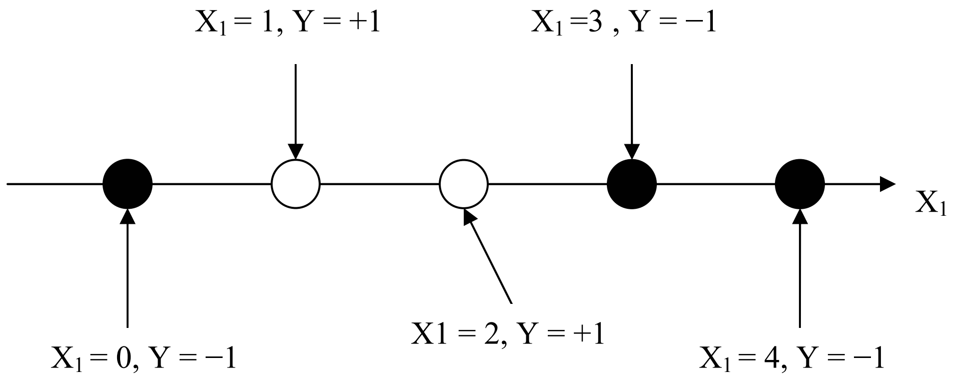

Figure 1: Training set of five examples with their desired classifications.

[a] Define a transformation (i.e., a feature mapping function) that maps the data into $\mathbf{R}^2$ (i.e., a function, F, that maps a data point X_1 from $\mathbf{R}^1$ to $(X'_1, X'_2) = F(X_1)$ in $\mathbf{R}^2$) such that the data becomes linearly separable.

[b] Construct a maximum-margin separating hyperplane. This hyperplane will be a line in $\mathbf{R}^2$, which can be parameterized by its normal equation, i.e., $w_1X'_1 + w_2X'_2 + c = 0$ for appropriate choices of w_1, w_2, c and all points X'_1, X'_2 on the plane. Here, $(X'_1, X'_2) = F(X_1)$ is the result from applying the function F you developed in [a] to the original feature X_1. Also, explicitly compute the margin for your classifier. Circle the support vectors.

Question 5: Face Detection using Neural Networks

Implement a feed-forward neural network with one hidden layer, using the back-propagation algorithm to train this network to detect the existence of a human face in a grayscale image. To keep things simple so you can concentrate on the neural network implementation, you will not do any preprocessing of the images. This is a simplistic approach, but it demonstrates how much a neural network can do even in cases where a task is not well understood or where it's hard to use domain knowledge to pre-compute important features to help the learning algorithm.

As always, you may discuss general issues with the professor, the TA or other students. However, the implementation must be entirely your own. In particular, you are not to look at other code related to back-propagation that you might find on the web or elsewhere, and you may not share code with other students in the class.

Part 1: Implement the Back-Propagation Algorithm

Your primary task is to implement the back-propagation algorithm given in Figure 20.25 in the textbook. Your implementation should represent a network with a single hidden layer.

The learning problem will be specified to your algorithm using the following two components:

1. an array of examples, each of which will be a long array of doubles corresponding to the image's pixel intensity values in row-major order
2. an array of corresponding (string) classifications (there are only 2 possible classifications, indicating if the given image contains a face or not)

The number of inputs to the neural network is the number of pixels in the input image. Initially, the hidden layer should have **10** hidden units, although you will experiment with this parameter. Throughout the assignment you should assume full network-connectivity, which means that every input unit should be connected to every hidden unit, and every hidden unit should be connected to every output unit.

Each unit in the hidden layer and the output layer should compute the sigmoid function. This function will return a real value in the range [0..1], not a binary value as the linear threshold unit does. Initialize all weights in the network to random values in the range [-0.5, 0.5]. You may experiment with values for the learning rate parameter, α, but initially try setting this parameter to 0.3. *Implement all real-valued objects, including activation values and weights, as doubles.*

The goal is to train the network so that it correctly classifies examples based on the input pixel values. When you classify an example, you should classify it based on the highest-valued output unit. You should have one output unit for a positive classification (positive detection of a human face) and one output unit for a negative classification (non-detection of a human face). You can think of this as each output unit learning to recognize its particular class of training examples. The target output should be [0.9, 0.1] when the correct classification of an image is the positive detection of a face. In general, one output unit's target value will be set to 0.9 and the other will be set to 0.1. These target values should be used instead of the more obvious 0's and 1's to prevent unstable weight behavior, as the sigmoid function only reaches 1 or 0 at $+\infty$ and $-\infty$.

You can implement this program any way you like, though you may want to implement routines called `ForwardProp`, `BackProp`, `Train`, and `Test`, among others. Additionally, you should call your program from the command line using the following command.

```
java HW4 <image file directory> <training images list file>
<testing images list file> <learning rate> <number of hidden units>
<number of training epochs>
```

There's more information about list files and the image directory in Part2. **Important: Please make sure your main class is named "HW4" (case sensitive) so as to make our testing easier.**

In summary, your program should do the following:

1. Determine the number of input and output units based on the problem specification.
2. Build and train a single-hidden-layer artificial neural network, using the training examples and back-propagation to learn the weights. Run a number of epochs (as given below).
3. Classify each testing example from <testing images list file> using the learned network.
4. Report the test set accuracy rate, and list for each incorrectly classified example the file name, the target value as specified in the test data file, as well as the value computed by your network. Do not print the input image's intensity vector because it is too big.

If the pseudo-code notation for the back-propagation algorithm given in the book in Figure 20.25 is confusing to you, consider the following alternative definition. Using the notation in Figure 20.24, the back-propagation algorithm for a network with one hidden layer where the output units are referenced using *i*, the hidden units are referenced using *j*, and the input units are referenced using *k*, is given below. Hence, *a_k* refers to an input example's values for each input unit, and *y_i* specifies the example's target (a.k.a. teacher) output values for each output unit.

```
in_j = $\text{sum}_k$ (w_k, j * a_k)                  // Forward pass starts. Compute weighed
                                                      // input to all hidden units.
a_j = sigmoid(in_j)                                   // Compute outputs at all hidden units
in_i = $\text{sum}_j$(w_j, i * a_j)                   // Compute weighed inputs to all output
                                                      // units
a_i = sigmoid(in_i)                                   // Compute outputs at all output units
del_i = a_i * (1 − a_i) * (y_i – a_i)                 // Backward pass starts.
                                                      // Compute "modified error" at output units
del_j = a_j * (1 – a_j) * $\text{sum}_i$(w_j, i * del_i)  // Compute "modified error" at all hidden
                                                      // units
w_j, i = w_j, i + (alpha * a_j * del_i)               // update weights between hidden and output
                                                      // units
w_k,j = w_k,j + (alpha * a_k * del_j)                 // update weights between input and hidden
                                                      // units
```

Part 2: Training Neural Nets to Detect Faces

The Data

The experiments in this part will focus on the task of detecting the existence of human faces in grayscale images. The images used in this study were created by the MIT Center for Biological and Computation Learning. The original CBCL Face Database #1's training set has 2,429 faces and 4,548 non-faces, and its testing set has 472 faces and 23,573 non-faces. A subset of the training set and testing set will be used for this assignment. The images' resolution has been reduced so they are all of size 19 × 19 pixels in order to keep training times manageable while still retaining enough information for good classification performance. Consequently, your neural net will *always* have 361 (= 19 × 19) input units. For the face detection learning task, it will have 2 output units.

To get started, copy the following four directories to your own working directory from the directory `/p/course/cs540-dyer/public/html/spring07/hw/hw4/` Be sure to keep the directory names as given.

File Set	**Directory in ~dyer/cs540/hw/hw4/**
Training Images with Faces	train-face
Training Images without Faces	train-non-face
Testing Images with Faces	test-face
Testing Images without Faces	test-non-face

The name of your working directory will be the first argument that you specify on the command line to run your program. PGM images can be viewed using any image viewer.

On the Linux machines a program called `xv` is an easy one to use to view the input images.

A Java class is provided to collect information from a set of PGM images and present it in a format ready for neural network training. Please read the comments in the code, including example usage in the header. The required input is a top-level image directory and the image list file. It returns images in linear arrays in row-major order, with intensity values in range [-1..1]. It also parses the filename and returns the classification of each image as a string. You may instead write your own code for handling the images if you wish, but make sure it uses the same image list file format.

This program requires a file containing the list of all images files that you want processed. We provide four files for testing your homework. For Part 2, use the set of images in the two list files: *train.list* for training and *test.list* for testing. These lists contain 120 and 40 images, respectively.

Also, if you want to write your own code for using the PGM images for training, information about the format can be found on the web; see link on homework page.

Be aware that the training and testing sets contain many examples. Although this program should run in a reasonable amount of time on most machines, you should debug your program by using only a small subset of the training data.

Training

After you have implemented your neural network, you should train it using the training list given above. You should run at least 300 epochs, and after every 10 epochs compute the sum of squared errors (SSE) on the training set, and the accuracy (i.e., percentage correct classification) on the testing set. ***Make sure your program is not using the testing examples for learning!*** Compute SSE as follows:

$$SSE = \sum_{j=1}^{m} 1/2 \sum_{i=0}^{n-1} (T_{ij} - O_{ij})^2$$

where

- *m* is the number of examples in the training set,
- *n* is the number of output units,
- T_{ij} is the target value (either 0.1 or 0.9) of the *i*th output unit for the *j*th training example, and
- O_{ij} is the actual real-valued output of the *i*th output unit for the *j*th training example.

You should compute this SSE value after every 10th epoch by stopping back-propagation and running through all of the training examples to compute this error value. You should not compute this "on the fly" after each training example is used to update the weights because then the error for each example would be based on a different network (i.e., set of weights).

We will not be auto-grading this section, so the `HW4.java` code you submit need not print the SSE information, and it does not need to print the test set accuracy information until the end.

Plot the SSE data as follows. The *x*-axis will show the epoch number and the *y*-axis will show the SSE value. Plot a point every 10 epochs. Similarly, plot the test set accuracy as a function of the number of training epochs. Plot a point every 10 epochs.

You can create these plots manually or by entering the data into any plotting program (e.g. in Matlab, Gnuplot, Excel). **Be sure to label all axes in your plots!** *Hint: If you print this information to a file in the correct format, you should be able to import it into one of these programs without retyping.*

Based on your test set accuracy, and the incorrectly-classified test examples, briefly discuss the performance of your program. Plot the two curves (SSE and accuracy). Is there any evidence of overfitting? What number of training epochs produces optimal results? View each incorrectly-classified image. Do the neural network's misclassifications also appear relatively similar to the human eye? Include a printout of at least one such example.

Experiment with Parameter Settings

Repeat the training and testing steps given in Part 2 after varying the number of hidden units in your network. Use at least value of **5**, **10**, and **20**. Show the training plot and the final test set accuracy for each of these new networks, using a separate graph for each. Comment on how performance changes with the number of hidden units. Discuss how the number of hidden units affects classification accuracy on the test set. Compare the number of each epochs that seem necessary for the network to "converge", and qualitatively compare the runtimes for each number of hidden units.

Repeat procedure to test for different values of the learning rate parameter, α, Trying at least values **0.1**, **0.3**, and **0.5**. Again, submit the plots. In total you should produce at least 9 plots for all possible combinations of hidden units and α. Specify what you consider to be the "best" set of parameters (α, number of hidden units, and number of epochs) for this problem based on your experiments, and briefly explain why.

Extra Credit (up to 10 points)

For extra credit, extend the ability of your program to detect all faces in a single large image. You can assume the size of the faces is fixed, but the positions are arbitrary. You'll have to extract windows centered on each pixel, reduce and normalize the images appropriately, and then run the neural network on each. The directory `~dyer/cs540/hw/hw3/extra-credit/` contains some sample test images.

What to Turn In

Electronically:

Copy your `.java` files into your `handin` directory using:

```
handin –c cs540-1 –a hw4 –d hw4-handin
```

where `HW4.java` is in a directory called `hw4-handin`.

Paper:

Hand in in class a hard copy of your answers to questions 1, 2, 3 and 4, and, for question 5, a report that includes the required plots, results (test set accuracy) of all your test runs, answers to discussion questions and image comparisons. Put this all together, stapled, with your name, login and the date on the top of the first page.