Artificial Intelligence
Fall 2018
Project #2

You are going to write two programs implementing the training and testing, respectively, of a neural network. The training of the network should use back propagation and be based on the pseudo code from Figure 18.24 in the textbook. I have added my own corrections and clarifications to this pseudo code as discussed in class. (I will post my annotated version of the pseudo code on the class home page along with the assignment.)

I am instituting two simplifications to the pseudo code. The first of these simplifications is that I am guaranteeing that *all neural networks will contain exactly one hidden layer*. As mentioned in class, a neural network with a single hidden layer can represent any continuous function, and using a single hidden layer has been very common in practice. The number of inputs in the network, the number of hidden nodes in the single hidden layer, and the number of output nodes will vary from network to network, as will the initial weights. The expected outputs for training and testing examples will always be 1 or 0, respectively indicating inclusion or exclusion from some Boolean class. (The second simplification to the pseudo code will be described later.)

The training and the testing programs that you create will rely on text files specifying neural networks. Each such text file might represent a neural network that has already been trained based on specific data, or it might represent an untrained network with initial weights that have been either manually configured or randomly generated. Your code should not randomly generate weights, so this is another change from the pseudo code in Figure 18.24 (not one of the two simplifications). The text files specifying neural networks will have the following format:

- The first line will contain three integers, separated by single spaces, representing the number of input nodes (N_i), the number of hidden nodes (N_h), and the number of output nodes (N_o).
- The next N_h lines specify the weights of edges pointing from input nodes to hidden nodes. The first of these lines specifies the weights of edges entering the first hidden node; the second line specifies the weights of edges entering the second hidden node; etc. Each of these N_h lines specifies $N_i + 1$ weights, which will be floating-point numbers separated by single spaces. These weights include the bias weight which is attached to a fixed input that always has its activation set to -1. (*Note that the fixed input should be -1, not +1, so we are not following the convention of the current edition of the textbook.*) Using a fixed input of -1 makes the bias weight equivalent to a threshold to which the total of the true weighted input can be compared. For each hidden node, the first weight represents the bias weight, and the next N_i weights represent the weights of edges from the input nodes to the hidden node.
- The next N_o lines specify the weights of edges pointing from hidden nodes to output nodes. The first of these lines specifies the weights of edges entering the first output node; the second line specifies the weights of edges entering the second output node; etc. Each of these N_o lines specifies $N_h + 1$ weights, which will be floating-point numbers separated by single spaces. These weights include the bias weight which is attached to a fixed input that always has its activation set to -1. For each output node, the first weight represents the bias weight, and the next N_h weights represent the weights of edges from the hidden nodes to the output node.

When your neural network training program is executed, it should prompt the user for the names of three text files representing the initial neural network, a training set, and an output file; one positive integer representing the number of epochs; and one floating-point value representing the learning rate. The first text file should contain the representation of the neural network before training (i.e., it will specify the size of each layer and the initial weights of the network using the format that has previously been described). The second text file specifies the training set for the neural network; i.e., this file contains training examples with which to train the network. The first line of this file contains three integers separated by spaces: the number of training examples, N_i, and N_o (I guarantee that N_i and N_o will match the neural network being trained). Every other line of the file specifies a single example and contains N_i floating-point inputs (the values of the example's input attributes) followed by N_o Boolean outputs (each is 0 or 1). The number of epochs specifies how many times the outer loop of the pseudo code (the *repeat...until* loop in the pseudo code) should iterate. This is the second simplification to the given pseudo code; instead of training until some stopping criterion is satisfied, *training will proceed for a specified number of epochs*. (Remember that each epoch loops through the entire training set, updating the weights of the network for every training example.) The learning rate is a floating-point value that determines how fast weights are updated; values that are two big will cause weights to overshoot their appropriate values, and values that are too low will cause the learning to be inefficient, requiring too many epochs before reaching a near-optimal state. The third text file, to be created by the training program, should have exactly the same format as the first text file, but the weights contained in this file should be the learned weights of the trained neural network.

When your neural network testing program is executed, it should prompt the user for the names of three text files. The first text file contains the representation of a neural network (presumably one that has already been trained) using the previously described format. The second text file specifies the test set for the neural network; i.e., this file contains testing examples with which to test the network. The format of this file is the same as the format for the file that specifies a training set for a neural network (as has been described above). The program should iterate through every test example, and for each, it should compute the outputs according to the specified neural network and compare them to the expected outputs. When comparing actual outputs to expected outputs during testing, all actual outputs (i.e., the activations of the output nodes) that are ≥ 0.5 should be rounded up to 1, and all outputs that are < 0.5 should be rounded down to 0. (Note that *this rounding should not occur during training* when computing errors of output nodes. The rounding only occurs during testing when determining whether or not an output of the neural network is correct.) As testing proceeds, your program will need to keep track of various counts in order to compute several metrics that will soon be explained. The third text file should store the results of these metrics with a format that will soon be specified.

When computing activations of all non-input nodes in a neural network, *both executables should use a sigmoid function* as the activation function. This means that your training executable will need to compute the derivative the sigmoid function with respect to the input for each node when it is updating weights. Fortunately, this turns out to be very simple. The equation for the sigmoid function is: $sig(x) = 1 / (1 + e^{-x})$. The equation for the derivative of the sigmoid function with respect to its input is: $sig'(x) = sig(x) * (1 - sig(x))$.

For each Boolean output class or category, you can imagine a confusion matrix (a.k.a. contingency table) indicating four counts:

	Expected = 1	Expected = 0
Predicted = 1	A	B
Predicted = 0	C	D

Here, A is the number of test examples that are correctly placed in the class; B is the number of test examples that are predicted to be belong to the class even though they do not really belong to the class; C is the number of test examples that are not predicted to belong to the class even though they do belong to the class; and D is the number of test examples that are correctly predicted not to belong to the class. Based on these counts, your program should compute the following metrics for each class or category:

(1) *Overall accuracy* = (A + D) / (A + B + C + D); this is the fraction of examples that are correctly predicted with respect to the current class. Overall accuracy is generally considered a poor evaluation metric for Boolean classification tasks. If most examples do not belong to most classes, a system can achieve a high overall accuracy by trivially predicting that all examples belong to no classes.
(2) *Precision* = A / (A + B); of those examples which are predicted to belong to the current class, this is the fraction that actually belong to the class.
(3) *Recall* = A / (A + C); of those examples which actually belong to the current class, this is the fraction that are predicted to belong to the class.
(4) *F1* = (2 * Precision * Recall) / (Precision + Recall); this is a way of combining precision and recall into a single metric, and it will always have a value in between the precision and the recall, closer to the lower of the two.

In addition to computing these metrics for every class, you can also compute each metric for the entire dataset. There are two ways to do this; one is called *micro-averaging*, which weights every decision equally, and the other is called *macro-averaging*, which weights every class equally. To compute the micro-averages for the first three metrics, you combine all of the contingency tables (for all classes) by adding all of the individual counts to obtain global counts for A, B, C, and D. You then apply the formulas given above for overall accuracy, precision, and recall based on the global counts; the formula for F1 remains the same and is based on the micro-averaged values of precision and recall. To compute macro-averages for the first three metrics, you average the values of overall accuracy, precision, or recall for the individual categories; the formula for F1 again remains the same and is based on the macro-averaged values of precision and recall.

The results text file created by your testing program should contain $N_o + 2$ lines. Each of the first N_o lines corresponds to a single category and should contain the eight values computed for A, B, C, D, and the four metrics discussed above, separated by single spaces. The remaining two lines should contain the four values computed for the micro-averaged metrics and the four values computed for the macro-averaged metrics, respectively. Note that if N_o is 1 (i.e., there is only a single output node), then all three lines will display the same values for the four metrics.

I am going to provide you with two datasets to test your neural network. The first dataset is the Wisconsin diagnosis breast cancer (WDBC) dataset, which I found on-line. Every example in the dataset represents a patient, and in the raw data file, each example includes 31 input attributes and one output. The input attributes include a patient ID and 30 real-valued features that have been computed as follows: (1) First, ten real-valued features have been computed for each cell nucleus; namely, (a) radius (mean of distances from center to points on the perimeter), (b) texture (standard deviation of gray-scale values), (c) perimeter, (d) area, (e) smoothness (local variation in radius lengths), (f) compactness (perimeter^2 / area - 1.0), (g) concavity (severity of concave portions of the contour), (h) concave points (number of concave portions of the contour), (i) symmetry, and (j) fractal dimension ("coastline approximation" - 1). Note that I do not understand some of these features, but for our purposes, it doesn't matter. (2) Next, for each of these features, the mean, standard error, and worst values (actually, the mean of the three largest values) have been computed across all cells, thus leading to 30 real-valued features per patient. The output is either M (for malignant) or B (for benign). The raw data includes 569 examples; 357 are classified as benign, and 212 are classified as malignant. The raw data is available in a file called wdbc.data, which I will make available to you, but you don't need to use it. I will also provide a text file called wdbc.names, containing a description of the dataset by its creators. Google searches related to the dataset reveal much additional information.

I have pre-processed the raw data to make our task simpler for us. First, I have randomly divided the data into a training set with 300 examples (188 are classified as benign, 112 are classified as malignant) and a test set (that I am really using more like a tuning set) with 269 examples (169 are classified as benign, 100 are classified as malignant). I have stripped the patient IDs, since they are not useful for learning. I have moved the output to the end of each row, and have converted M to 0 and B to 1. I have normalized every other column by dividing each original value in the column by the maximum value in the column. (I have found that the neural network training works better this way compared to using the raw numbers, although there are likely better methods of normalization.) All in all, I have converted the files to formats that are appropriate for use with neural networks that satisfy the specifications of this assignment.

For this dataset, I have found that a using neural network consisting of a single hidden layer with 5 hidden nodes, a learning rate of 0.1, and 100 epochs of training leads to pretty good results. I will provide you with my initial neural network file; the initial weights are pseudo-randomly generated fractions between 0 and 1. I will also provide you with my trained neural network file and my results file for the suggested combination of parameters. This will allow you to test your training and testing programs against my data. You will note that the overall accuracy is over 96%, and the F1 metric is over 97%. Since there is only one category being predicted, it turns out that the micro-averaged results and the macro-averaged metrics will exactly match the metrics for the single category; thus, the results look somewhat redundant for this dataset.

You should *use double precision floating-point variables* to represent all floating-point values. All output files (networks with trained weights and results) should *display all floating-point values with exactly three digits after the decimal point*, including trailing zeroes. If you follow all instructions, *your program's output files should be identical to mine*. If you see any values that are different than mine, this is an indication of a bug (either with the formatting or with the algorithm itself), even if the final results are as good as, or better than, those of my program.

To allow you to test your programs on a dataset with multiple output categories, I have also created a contrived second dataset. Each example represents a student, and includes five input attributes indicating scores on five assignments (each has been normalized to be a fraction between 0 and 1), and four output attributes representing categories of grades (D or better, C or better, B or better, and A, based on strict cutoffs of .60, .70, .80, and .90, respectively). Of course, these output categories, which are determined by the average of the students' scores, are not independent of each other, but your neural network program won't know that! It is conceivable, for example, that your trained network might predict that an example belongs to the B or better category, but not the C or better category; however, I am not seeing anything like that happen after a reasonable number of training epochs with decently configured parameters. I have created a randomly generated training set with 300 examples and a randomly generated test set with 100 examples. (The distributions of scores are not uniform.)

For this dataset, I have found that a neural network consisting of a single hidden layer with 10 hidden nodes, a learning rate of 0.05, and 100 epochs of training lead to pretty good results; 9 failing students are predicted to be in the D or better category, and 3 students who should receive an A are not predicted to be in the A category, but everything else is correct. The results are even better after 500 epochs of training; 3 failing students are predicted to be in the D or better category, but everything else is correct. I am going to provide you with my initial neural network file (including pseudo-randomly generated initial weights between 0 and 1), my trained neural network file after 100 epochs, and my results file based on this trained network.

There is one more component to this project. *I want every student to either find or create an interesting dataset* that can be used to train and test neural networks adhering to the specifications of this assignment. If you have followed all of the instructions correctly, the results of my neural network should exactly match those of yours if trained using the same parameters. Therefore, when you submit your project, also send me the following:

- Your dataset, including a training set file and a test set file;
- A description of the data (including an explanation of the input attributes and the outputs);
- A combination of parameters that leads to reasonable learning, if possible (including a learning rate, the number of hidden nodes in the hidden layer, and the number of epochs);
- Text files representing your original untrained neural network, your trained network using the specified parameters, and your results file;
- A description of how the initial weights of your neural network have been generated;
- If you have created the dataset yourself, or if you have modified a dataset that you have found, a description of how you have created or modified it; and,
- If you have found the dataset, an explanation of where it came from.

In general, you can choose any programming language as long as I can easily compile and run your program. However, I am adding a few additional constraints:

- Do not use MATLAB, Octave, or any similar language. (If you are unsure, ask me.)
- Do not use any linear algebra libraries or libraries related to neural networks.
- Perform calculations for individual nodes and weights (i.e., do not use vector computations or matrix operations to calculate weight adjustments).

The project is due before midnight on the night of *Sunday, December 9*. After you have completed the assignment, e-mail me (*CarlSable.Cooper@gmail.com*) your code, simple instructions on how to compile and run your training and testing programs, and your own dataset including all the components described previously. Make sure that you have read all the requirements carefully and have not forgotten anything. Ask me if you have any questions.