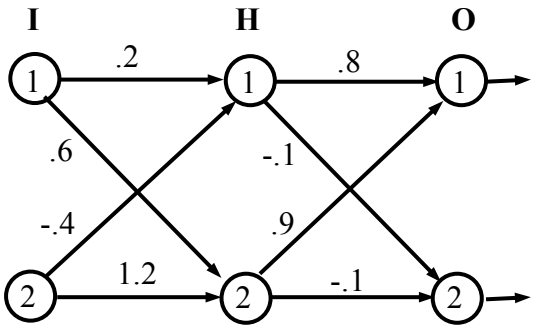
CMSC 422: Assignment 2 – Error Backpropagation – Spring 2020

The purpose of this assignment is to help you gain familiarity with basic error backpropagation, and more generally, with the concept of using gradient descent as the basis for learning.

1. Simple Numerical Computation



Consider the layered neural network shown at the right having input nodes I1 and I2, hidden nodes H1 and H2, and output nodes O1 and O2 (node numbers are written inside the nodes). The activation level a_j of H/O node j is computed using the usual logistic function (s = 1) where

$$in_j = \sum_i w_{ji} a_i$$

is the input to node j. Suppose that basic sequential error backpropagation with momentum is used for training, with learning rate η = 0.1 and momentum coefficient α = 0.5. The current state of the network's weights after the previous iteration of learning is shown in the picture, with weights next to connections (and with biases not being shown). For the current step of learning at iteration t, the inputs are a_{I1} = 1.0 and a_{I2} = -1.0, while c_{O1} = 1.0 and c_{O2} = 0.0 are the target/correct output values. Assume that with this input pattern, during the forward pass of iteration t, a_{H1} = .7, a_{H2} = .3, a_{O1} = .8, and a_{O2} = .4 are the actual activation levels produced (i.e., *assume* these are correct values with the nodes' current bias values and use them in answering the questions below). E is the usual squared error here. Compute what the following values will be at the end of learning during iteration t:

a. Compute the output node delta values δ_{O1} and δ_{O2}.

b. What is the hidden node delta value δ_{H1} assuming that delta values are propagated backwards *before* weight changes are made?

c. What is the new value of weight w_{O1H2} between hidden node H2 and output node O1 after this iteration, where Δw_{O1H2} was 0.001 on the previous time step?

2. Gradient Descent

Consider a feedforward neural network with n input nodes that are fully connected to m output nodes; there are no hidden nodes and no bias units. Output unit k computes its output as

$$o_k = 1 - e^{-in_k}$$

and its input as $in_k = \frac{1}{2}\sum_l (w_{kl} - x_l)^2$ where $\vec{w}_k$ is its incoming weight vector, l ranges over the input units, and $\vec{x}$ is the input vector. An error function for the d^{th} training example is the usual $E^d = \frac{1}{2}\sum_k (t_k - o_k)^2$ where k ranges over the output units and t_k is the target value.

a. Sketch a plot of o_k as a function of in_k without using a calculator/computer.

b. Use gradient descent to derive a learning rule $\Delta w_{ji} = \ldots$ based on the error measure above. Assume incremental gradient descent is being used so there is no need to sum over the training examples d. Express your learning rule *solely* as a function of local quantities t_j , o_j , x_i , w_{ji} and learning rate η.

c. Explain in a few sentences how this learning rule appropriately changes the weight w_{ji} when o_j is too small, too big, or just right for a given input pattern d so that o_j would be more correct were pattern d immediately used as an input again, assuming an arbitrarily small learning rate.

3. Classification with Error Backpropagation: Two Moons Data

In this problem, you will use the error backpropagation classifier in scikit-learn (https://scikit-learn.org/stable/documentation) to see how its performance compares to that of an elementary perceptron on the "two moons" data used in Assignment 1.

Create a python script file *moons.py* that first generates a data set of 500 samples by calling *make_moons(),* where you use the default parameter values except for assigning the same initial random seed that you used consistently in runs with the perceptron in Assignment 1; e.g., *random_state* = 13. You will again need to vary the amount of noise in the data by assigning the appropriate input parameter (*noise* = …) as described below. Once you create a data set, randomly partition it into stratified training (75% of data) and test (25%) data sets using *train_test_split()*, again, just as was done with the first assignment. Your script file should then print to a file named *moonsResults.txt* your name, the noise level and random seed used in generating the data, the exact number of training and test data examples produced by *train_test_split()*. Once the data has been generated, partitioned and displayed, your script file should continue on to train scikit-learn's error backpropagation classifier *MLPClassifier()* on the training data to classify examples correctly. Use a single hidden layer with six nodes, the hyperbolic tangent as the activation function, and *lbfgs* as the solver, but otherwise just use the classifier's defaults. Generate output to the same *moonsResults.txt* file that specifies the post-training accuracies on the training data and the test data.

a. Using your script file, do five independent runs of backpropagation learning of the two moons data. For each run, generate 500 samples using the same random seed but having a different value 0.01, 0.1, 0.2, 0.5, or 2.0 for the *noise* input parameter in each run (or, you can simply use these data sets from Assignment 1 if you still have them). Based on the results as described above, create and turn in (hardcopy) a five line table where the columns are (left to right): noise level, backprop accuracy on the training data, backprop accuracy on the test data, perceptron accuracy on the training data, and perceptron accuracy on the test data. The latter two columns should contain your results on the same data from Assignment 1 (i.e., you do not need to re-run the perceptron).

b. How did the error backpropagation's accuracy change as the noise level increased from near zero to 2.0? Explain why this occurred.

c. How did backpropagation's accuracy compare to that of the perceptron in general? Explain why this occurred in terms of the decision surfaces associated with these two approaches to learning.

4. Regression with Error Backpropagation: Diabetes Prognosis Data

Using perceptron and backpropagation learning with the two moons data involved pattern classification. In contrast, here we turn to a regression problem where, based on input examples, we learn to predict numerical output values rather than classes. Specifically, in this problem you will use the diabetes dataset that comes with scikit-learn (use *datasets.load_diabetes()*). Each of the 442 examples in this data set has 10 input features for a person (age, gender, body mass index, blood pressure, etc.) and a numerical score (the target value) that indicates how much that person's disease progressed one year later.

Use the error backpropagation (multilayer perceptron) regression module in scikit-learn to train a neural network to predict a person's progression score based on the input features. Specifically, create a python script file *diabetesBase.py* that first loads the diabetes data, and then partitions it into training and test sets using *train_test_split()*. Use the default parameter values for this, except be sure to give a specific value to its parameter *random_state*, e.g., *random_state* = 13, and to use this value consistently in your simulations to facilitate the comparisons described below. Your script file should then train a network using all of the default values for parameters, e.g., creating the network using

mlp = MLPRegressor(random_state = 13),

except for assigning an initial random seed (13 here) that you use consistently in all of your training runs to facilitate comparisons. Call this the baseline run. Your script file should create an output file named *diabetesBaseResults.txt* containing the following information roughly in this order:

- parameter/attribute values used, taken directly from the neural network object *mlp*, including random seed used, number of layers, number of nodes in hidden layer(s), and number of epochs of training;
- RMSE measured separately on both the training data and test data, both before training and after training; implement your own function *rmse()* in your script file *diabetesBase.py* and use it to measure these; for pre-training measures, you can use the results after one epoch;
- the target values for the test data, and the actual post-training output values produced by the network for the same test data; and
- the weights and biases arrays learned by the network.

a. State the RMSE values that you obtained with the baseline network for the training data and test data, both before training and after training.

b. Create a new python script file *diabetesBest.py* (e.g., initialize it as a copy of *diabetesBase.py*). Using the new file, try some experiments in which you vary the number of hidden units, the number of epochs of training used, and perhaps other parameters to improve the performance of the backpropagation learning on the test data. Be sure to use the same random seed as in (a). You do not need to exhaustively vary the parameters, but do enough variations to get a significant improvement, e.g., an RMSE on the post-training test data that is half or less of what you found with the baseline run in (a). Your script file *diabetesBest.py* should create an output file named *diabetesBestResults.txt* containing the same information in the same order as with the baseline run. Retain a version of *diabetesBest.py* with parameter values in place for the best results you obtained and the corresponding output file *diabetesBestResults.txt* to turn in.

c. Would you expect to get significantly improved performance, compared to what you observed in (b), if you standardized the input features in the given diabetes data? Why or why not?

d. Write and turn in a brief (one page or less) summary of your work in (a) and (b) above. This summary should include a small table giving the post-training RMSE values that you obtained in (a) and (b), where for (b) just give the results for the single best run you did. For (b) also summarize the parameter variations that you tried, which of these helped improve the results on the test data, which didn't, and what the best set of parameter values you found were.

What should I turn in?

The hardcopy and electronic submissions are due at different times. The hard copy portion is due at the start of class Thurs., Feb. 27, the electronic submission is due earlier at 11:30 pm Weds., Feb 26.

Hardcopy: Your answers to the questions a – c in Problems 1 – 3; and a, c, and d in Problem 4.

Electronic submission: For Problems 3 and 4, turn in a single zip file. For Problem 3, this zip file should include your script file *moons.py* and your output file *moonsResults.txt* (both just for the case where a noise level of 0.1 is present). For Problem 4, your zip file should include *diabetesBase.py, diabetesBaseResults.txt, diabetesBest.py, and diabetesBestResults.txt* . Use the Computer Science Department project submission server at https://submit.cs.umd.edu to submit the zip file.