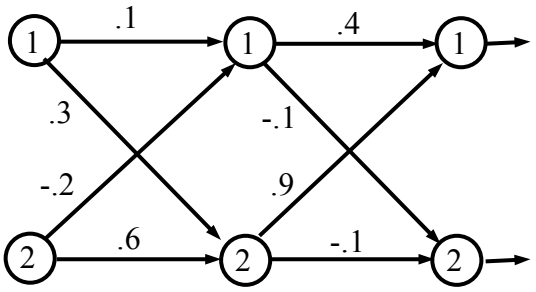
CMSC 422: Assignment 2 – Error Backpropagation – Spring 2019

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. 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 $in_j = \sum_{i \varepsilon I} w_{ji} a_i$ is the input to node j. Suppose that basic sequential error backpropagation with momentum is used for training, with learning rate $\eta = 2.0$ and momentum coefficient $\alpha = 0.8$. The current state of the network 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 $c_{O1} = 0.1$ and $c_{O2} = 0.9$ are the target/correct output values. Assume that with this input pattern, during the forward pass of iteration t, $a_{H1} = .4$, $a_{H2} = .5$, $a_{O1} = .6$, 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:

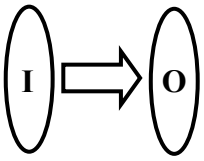


(a) Compute the output node delta values δ_{O1} and δ_{O2}.

(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 -0.1 on the previous time step?

2. Consider a single-layer, fully-connected neural network such as that pictured at right, where each input node in I connects to each output node in O, and where the real-valued input and activation levels of each output node k are given by the usual $in_{Ok} = \sum_{l \in I} w_{kl} a_{Il}$ and the equation $a_{Ok} = (1 + e^{-s_{Ok}\, in_{Ok}})^{-1}$ where w_{kl} are adaptable weights, and s_{Ok} is a "gain" that multiplies in_{Ok} in the exponent. Note that a_{Ok} is essentially the familiar logistic function except that s_{Ok} is no longer a single global constant, but instead is node-specific and can potentially change during learning just like weights, perhaps becoming negative. For a set of input patterns $\vec{a}_I^{\,p}$, it is desired to learn a set of corresponding output patterns that are unrestricted except that all of the output vectors are to be of unit length. In other words, training data $S = \{ \vec{a}_I^{\,p} \mid p = 1, 2, \ldots \}$ do *not* include associated "target" output patterns $\vec{c}^{\,p}$; all that is required is for the network to learn to produce an arbitrary unit length output $\vec{a}_{Ok}$ whenever a vector $\vec{a}_I^{\,p}$ in S is input to the network. Suppose that you decide to represent this constraint by an "error measure" (objective function) $E^p = \frac{1}{2}(1 - \sum_{k \in O} (a_{Ok}^p)^2)$ that is to be minimized by an iterative gradient descent procedure during network training. In this E^p equation, a_{Ok}^p refers to the actual activation level produced by the network for the k^{th} output node when $\vec{a}_I^{\,p}$ is the input pattern for the p^{th} training example.



(a) Use gradient descent with the given E^p to derive a learning rule for how s_{Oj} in this network should change during training after seeing the p^{th} training example (superscripts p may be omitted in this

derivation), expressing Δs_{Oj} solely in terms of a_{Oj}, a_{Ii} and/or in_{Oj}, and constant parameters. Clearly show each step in your derivation and circle your final result $\Delta s_{Oj} = \ldots$.

(b) Would the Δs_{Oj} derived in (a), when used with an appropriate weight change rule derived from the same objective function E^p using gradient descent, be expected to produce the desired unit length output vectors after training? Why or why not?

3. Using Error Backpropagation to Classify Radar Signals

Here you are to again train a neural network to classify radar signals, but this time using an error back propagation network with two layers rather than a single-layer perceptron.

Download the Python code for error back propagation (multi-layer perceptrons) from Marsland's ML textbook (http://stephenmonika.net). You are to train an error back propagation network using *mlp.py* to classify radar ping returns as to whether they are good (1) or bad (0). Use the same data set of radar returns as was used with the perceptron project, keeping the same ratios of training and testing data (80% of the data for training, 20% for testing). To enable a fair comparison to the perceptron results, again take every fifth example in the original data set to be testing data, using the remaining data as training data. Also, in generating the results and answers below, only report the results that you obtained for the best run that you do. You do not need to use early-stopping with validation data.

Specifically, train a backprop net (use a script file named *runbp.py*) on this classification task (1 = good radar return signal, 0 = bad radar return signal) using the raw data in the *radarData.txt* file. Do at least six runs to get the best post-training results that you can. In doing this you can vary the initial weights, learning rate, number of hidden units, weight initialization method, and number of iterations until termination to optimize your results. The network should print its results to a file (*Results.txt*). This file should start with the number of training and testing examples used, the learning rate, number of iterations until termination, number of hidden nodes, and any other run-specific information. It should then report the initial pre-training weights, confusion matrix, and fraction of all possible inputs that are classified correctly before training (separately for both the training and testing data), and subsequently the same information post-training.

Answer these questions based solely on the *best* post-training results that you got:

(a) What are the apparent and estimated true error rates for this backprop network?

(b) Based on the confusion matrix, what was the most common test data error made by the network?

(c) How did the estimated true error rate you obtained here for backpropagation compare to the results that you obtained using an elementary perceptron in the previous homework?

(d) How do you account for the difference that you observed in (c) in terms of decision surfaces?

4. Using Error Backpropagation to Classify Radar Signals: Weka

There are many software environments, or "toolboxes", that one can use to quickly apply off-the-shelf machine learning methods to data sets. Here we give you an introduction to Weka, and in the next problem we look at scikit-learn. Weka is one of the easiest environments to use.

In this problem you are to use Weka's module for error back propagation learning, which is called a *Multilayer Perceptron* in Weka, to train a neural network with the data in the provided file named *radarData.arff* (same as the *radarData.txt* data but with header information added to be compatible with Weka). When you activate the backpropagation learner in Weka a line of text starting with

MultilayerPerceptron –L 0.3 –M 0.2 –N 500 ...

(where -L = learning rate, -M = momentum, -N = training epochs) should be evident near the top of the window. Use 10-fold cross validation for this problem.

While the default parameter values for the multilayer perceptron are reasonable, you should do a number of runs in which you vary these some (e.g., larger or smaller momentum, running for 2000 epochs, ...) to get the best results you can. To change these parameters, just before starting training, click (or right-click if that does not work) on the line of text

MultilayerPerceptron –L 0.3 –M 0.2 –N 500 ...

in the Classify window. This should produce a pop-up window where you can change the training parameters. For example, if you increase momentum *M* to 0.5 and the number of training epochs *N* to 2000, then after clicking OK in the pop-up window, the same line of text should now read

MultilayerPerceptron –L 0.3 –M 0.5 –N 2000 ...

Cut-and-paste a copy of the complete, unmodified contents of the Classifier Output Window for the best results you obtain into the file *ResultsWeka.txt* that you will need to turn in. Answer these questions *about the best results* that you obtained using 10-fold cross validation:

a. How many hidden units are used in the default network constructed by Weka? How was that determined?

b. For the best run that you did, given only the results reported by Weka, approximately what future error rate (estimate of true error) for this default neural network would you predict, what is the most common specific error(s) made using this trained neural network, and how do these results compare with those you obtained in the immediately previous Problem 3? (Note that because of the way Weka reports results, you can only make very rough estimates here.)

5. Error Backpropagation for Regression Using scikit-learn and the Boston Housing Data

Scikit-learn is a widely used, python-based, software environment. To access the documentation for scikit-learn, go to its home page http://scikit-learn.org/stable/documentation where you can find its User Guide and instructions for downloading it (e.g., *pip* or *conda*). While we have so far focused on learning in problems involving classification, here we turn to a regression problem where, based on input examples, we learn to predict continuous numerical output values rather than classes.

Specifically, in this problem you are to use the Boston Housing Dataset that comes with scikit-learn (e.g., use *datasets.load_boston()*). The housing data set has historically been widely used in machine learning as a benchmark. From 13 input features (neighborhood crime rate, how old the houses are, adjacency to Charles River, etc.), your goal is to predict the value of a house, making this a regression rather than a classification problem. Once you load this data, you will need to partition 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.

a. Use the error backpropagation (multilayer perceptron) regression module in scikit-learn to train a neural network to predict a house's cost based on the housing data. Specifically, create a python script file *housesBase.py* that trains 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 your training runs to facilitate comparisons. Call this the baseline run. Your script file should create an output file named *housesBaseResults.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 *housesBase.py* and use it to measure these;
- the target values for the test data, and the actual output values produced by the network for the same test data; and
- the weights and biases arrays learned by the network.

b. Create a new python script file *housesBest.py* (e.g., initialize it as a copy of *housesBase.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 *housesBest.py* should create an output file named *housesBestResults.txt* containing the same information in the same order as with the baseline run. Retain a version of *housesBest.py* with parameter values in place for the best results you obtained and the corresponding output file *housesBestResults.txt* to turn in.

c. Finally, create a new file *housesScaled.py* that is initially identical to *housesBest.py*. Modify file *housesScaled.py* so that before doing any training the input features are standardized to each have roughly a mean of zero and a standard deviation of 1. Keeping the parameters used the same as in (b), run *housesScaled.py* and record the same results in an output file named *housesScaledResults.txt*.

d. Write a brief (one page or less) summary of your work in (a) – (c) above. This summary should include a small table giving the post-training RMSE values that you obtained in (a) – (c), 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 helped improve the results on the test data, which didn't, and what the best set of parameter values you found were. For (c), state whether scaling the data before learning made the results better, worse or roughly the same, and why that occurred.

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 Thursday Feb. 28, 2019, and the electronic submission is due earlier at 11:59 pm Wednesday Feb. 27..

Hardcopy: Your answers to the questions in problems 1, 2, 3a-d, 4ab, and 5d.

Electronic submission: For problems 3 - 5, turn in a single zip file that includes the script and result files (*runbp.py*, *Results.txt*, *ResultsWeka.txt*, *housesBase.py, housesBaseResults.txt, housesBest.py, housesBestResults.txt, housesScaled.py,* and *housesScaledResults.txt).* Be sure to include any files you wrote that are needed to make your code run, including files such as *mlp.py* if you made changes to them. Submit this zip file using the Computer Science Department project submission server at https://submit.cs.umd.edu as was done in the previous homework.