NEURAL NETWORKS

PART 1 IMPLEMENTATION

The code for implementing this homework is contained in hw5.py, hw5data.py, feedforward.py, and neuralnetwork.py. Output can be found in out.txt.

The training, testing, and utility classes such as neural nodes are contained within neuralnetwork.py. The feedforward network itself is implemented in feedforward.py. To make lookup more intuitive and easy, each node contains its own copy of weights coming into the node as well as out. This means that we have two copies of the same weight. These weights are both updated during the weight update process. Since the theta and x's are assigned accordingly, the weights will undergo the same mathematical operations and stay the same.

Our images are 32 x 30 pixels; thus, our input layer has 960 nodes, our hidden layer has 100 nodes, and finally our output has 1 node.

The homework prompt has indicated that the values of each image pixel is a grayscale float from 0 to 1. When I used python's opencv library to read the images, I found that the values were integers ranging from 0 to around 32 (it varies). Thus, during the data loading and preprocessing stage, I normalized the image values.

Additionally, the prompt has asked me to create initial weights as random numbers ranging from -1000 to 1000. During implementation, I realized that these weights would create overflow in python's exp() function, and give me node-x's that are only either 1 or 0. This is problematic, since during our propagation, theta is calculated as

$$\delta_i^{l-1} = x_i^{l-1}\left(1 - x_i^{l-1}\right)\sum w_j^2 \delta_j^2$$

The term $x_i^{l-1}\left(1 - x_i^{l-1}\right)$ creates a problem when all x's are 1 or 0. Because evaluating this will yield 0 no matter what. In this case, our nodes would almost never 'learn' by updating. Instead, I chose to initialize weights as a random float from -1 to 0.

I tested the neural network with the test set before training, and once more after training. This gives us a good measure of how much improvement our neural network gained from experience. We use the test metrics mean absolute error $\frac{1}{N}\sum|\hat{y} - y|$ and root mean square error $\sqrt{\frac{1}{N}\sum(\hat{y} - y)^2}$

Before training, the testing mean absolute error was around .445, and RMSE was .541.

After testing of 20 epochs (training takes a very long time, so I chose 20 for the time being…which runs in several minutes) the MAE and RMSE went down to .176 and .324.

These values are not always the same, since initial weights are created at random. However, it's clear that there were considerable improvements from training.

PART 2 SOFTWARE FAMILIARIZATION

I first began by transforming the image inputs into tables so that other pre-programmed packages can easily interpret the data. These are stored in wekadata.csv and wekadatatest.csv. I imported these files into Weka, and used its MultiLayerPerceptron algorithm, but did not obtain very worthwhile results. Its mean absolute error and root-mean-square error were both around .5, which means that it's no better than an un-trained network.

Instead, I decided to download a python packaged called ffnet. This is a package specifically tailored towards feed-forward networks, and is accessible through python, although some of its code is written in fortran. To see my code to import data and test ffnet, please refer to ffnet.py.

I ran the same input data and test data using ffnet's train_momentum() function. It seems that many 'back-propagation' learning algorithms (including weka as well as ffnet) use a momentum term in addition to the learning rate, which is a rate multiplied by delta of weights in one layer previous of the weights we are currently considering. The output can be seen in ffnet-momentum-out.txt. You can see that the backpropagation network did not do what is expected, and somehow, all outputs were the same…

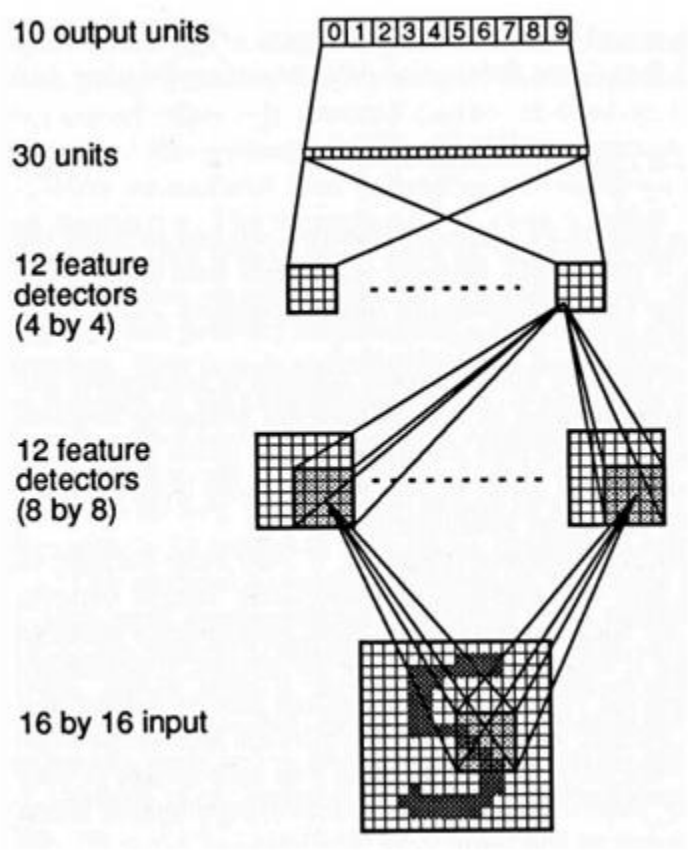
I noticed, however, that there were many options to train a network, even specifically just for feed-forward. Within the ffnet package, the options are BFGS, Conjugate Gradient, Genetic, Momentum (backpropagation), Rprop, and TNC. Since one of its examples used the train_tnc() learning algorithm, I tried it with this, and the output is shown in ffnet-tnc-out.txt.

Fortunately, this test yielded much better results, where line-by-line, we can see that the prediction was rather close to the target answer.

In general, however, I noticed that with our input (960-100-1) the network is quite large. My own program, Weka, and ffnet which are implemented primarily in python, java, and Fortran respectively, all ran rather slowly.

APPLICATION

During my research for scribe notes, I saw various interesting ways neural networks are used to identify and predict time series and patterns. It's interesting that this homework is very like one of the applications I found for hand-written digit recognition, which has an architecture like the following:



From: Introduction to the Theory of Neural Computation, Hertz, Krogh & Palmer, Addison-Wesley, 1991.

In addition, I saw some interesting examples online of using a feed-forward neural network to learn from a set of words from various languages, and can correctly guess what language any given word came from.