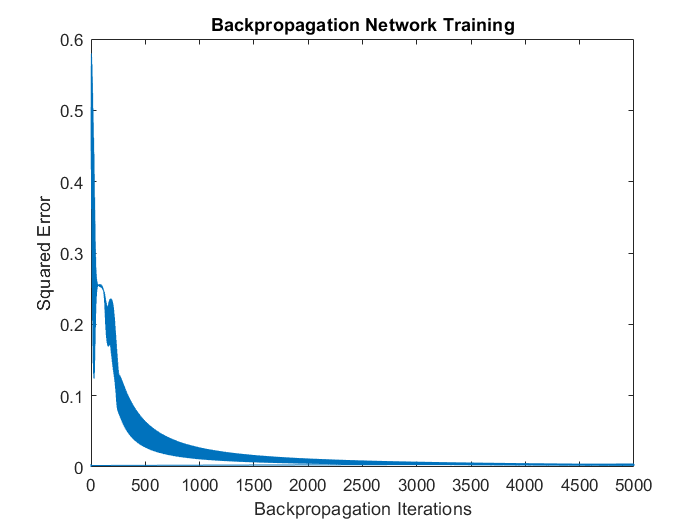
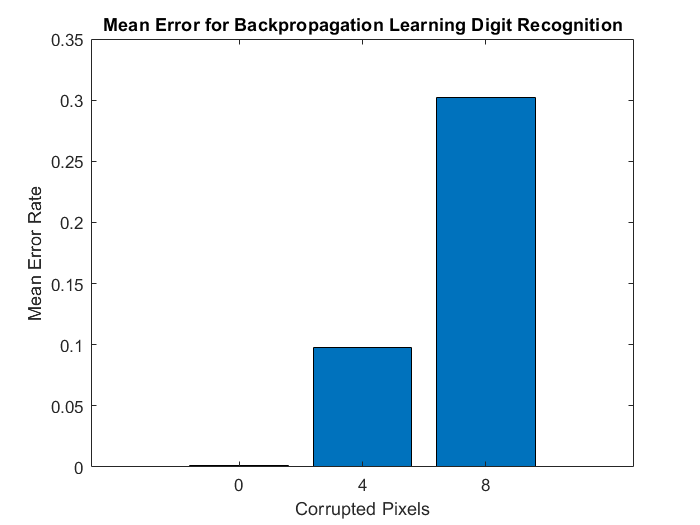
Peter Stanton

CSS 485

11/14/2017

Assignment 4

This assignment involves creating a backpropagation network for digit recognition, training, and applying normal and corrupted input to test accuracy. We train the network on clean input as shown in Figure 1, by iterating through given inputs. I found that iterating through the inputs sequentially rather than randomly selecting one for training significantly reduced the noise in my data.

This is not a universally seen form. Due to the randomness involved in the process, other shapes can be seen, including scenarios where convergence never occurs. Generally, though, this shape is representative, and learning usually converges around epoch 1000 or a little over that. Once trained, the network is then tested on input, as shown in Figure 2.

Note that pixel corruption is cumulative. Thus, for the 8-corrupted pixel test, an additional 4 were corrupted after the corrupted 4 tests. Code is commented out for another 8 pixels so you can run the script with a total of 12 corrupted pixels if you wish. Note that these results can also vary, but they tend to average out around the 0.5 range, though I’ve had runs with virtually no errors to runs with errors over 1.0. The worst runs usually coincide with a lack of convergence in training in the training performance graph. Another common result is for the 4 corrupt pixels to not significantly outperform the 8 corrupt pixels. Note that the error displayed here is calculated by sum(myError.^2)/length(myError), where myError is a 3x1 vector calculated from the outputs of the output layer neurons.

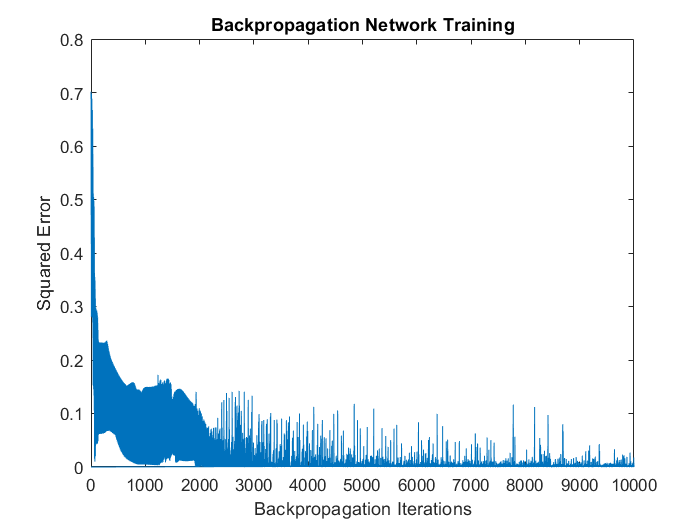
I found I had the best results when training and running for a set number of epochs rather than specifically testing for a threshold performance. I had issues with oscillation, and found that I would very often get false positives through oscillation.

Code follows:

Extra Credit:

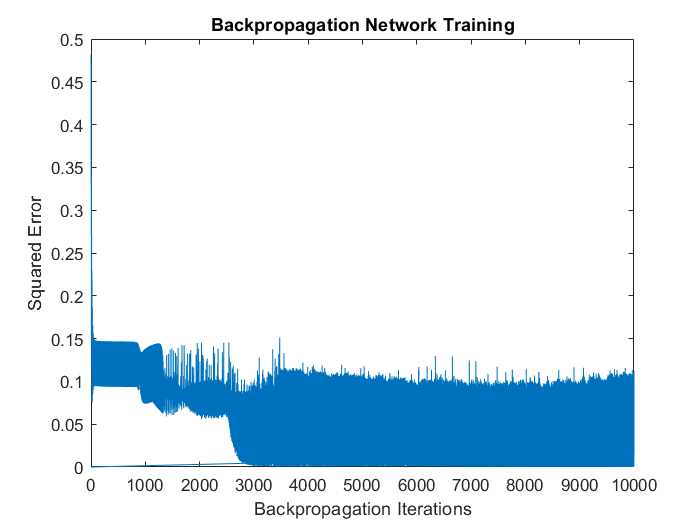
This only requires a straightforward adaptation of code. Credit to Tim for volunteering his input and output vectors for use. Really, all I had to do was copy my code over, and adjust a few matrix dimensions, and fiddle with learning rate and hidden layer neuron count a bit, and I get pretty similar results to Part 1.

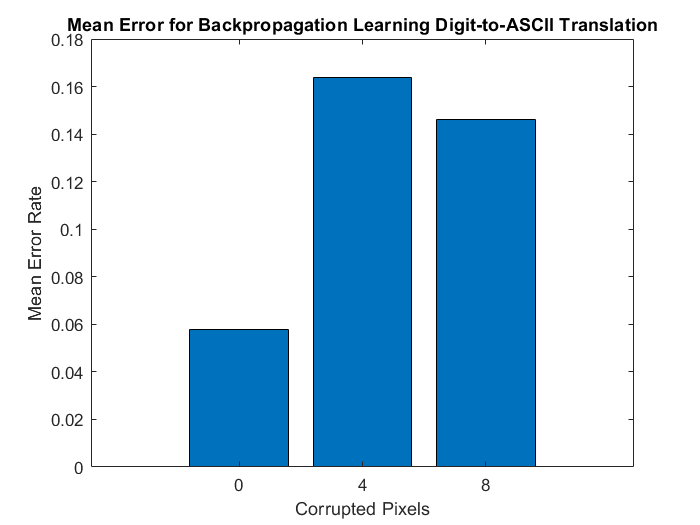
After much fiddling, I did turn up the learning rate to 0.1, and increased the number of neurons to 9, which seems to give me the best results, as shown below:



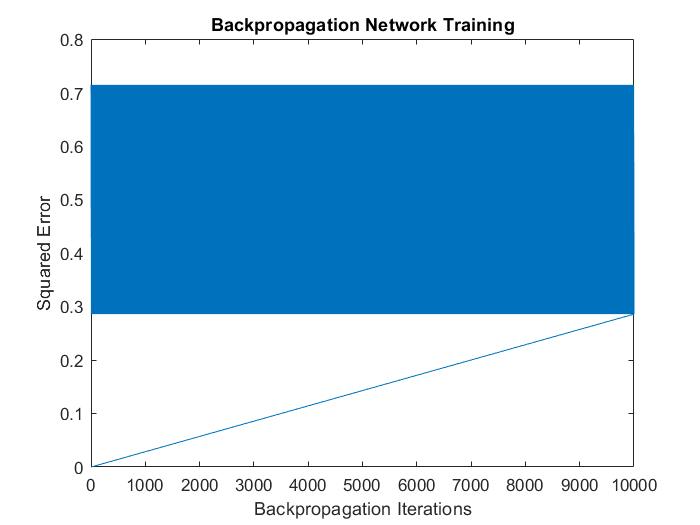


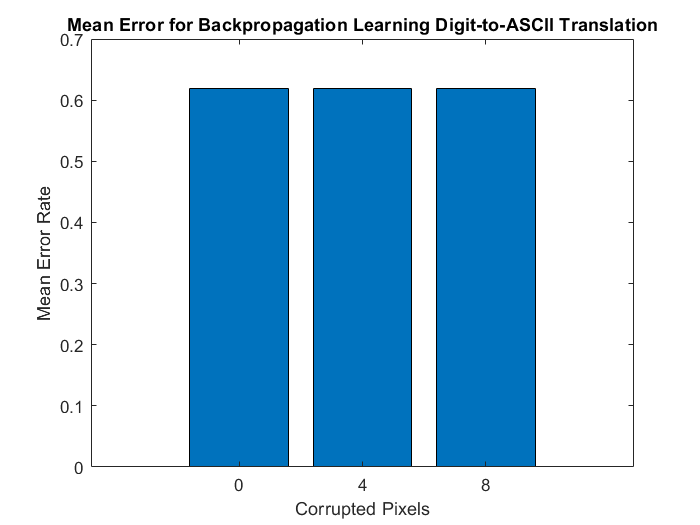
Again, figures run-to-run vary, sometimes significantly, but this is the general trend of results. I’m noticing that it seems more neurons are needed for training sets with lots of inputs. Learning rate of course obviously encourages the data to converge by magnifying changes to weights and biases in each epoch of backpropagation. Here, for example, is this network trained on two neurons:





Convergence no longer occurs, and accuracy is significantly degraded. Conversely, with too many neurons, like 90, we have a similar problem:





The network has seemingly become too large for the training set to effectively train.