The program was run in MATLAB. The user runs the main program 'neural_net_driver.m' by typing 'neural_net_driver' in the MATLAB console window. The program constructs, learns, and runs a neural network on a given dataset. The neural network is constructed and run in 'neural_net.m.' The program 'neural_net.m' makes a call to 'g_func.m,' which computes the function representing the output function of a given perceptron in the network. The neural network is 'learned' through the backpropogation algorithm, which is called in 'backpropogation.m.' The program 'backpropogation.m' makes a call to 'g_func_prime.m,' which computes the function representing the derivative of 'g_func.m.'

The program takes in a configuration file, a training set data file, and a testing set datafile, and the program outputs the sum-squared errors with respect to the training data and testing data over the learning iterations. In the end the program also outputs the learned output vectors over the training data set and the testing data set. In the neural network, the input vectors are fed into a network of perceptrons, which produce the output vectors. The equations for the perceptrons are give as below:

$$net = \sum_i w_i x_i \tag{1}$$

$$g(net) = \frac{a_{max} - a_{min}}{1 + e^{-net}} + a_{min} \tag{2}$$

where the x's represent the inputs for the preceptrons, the w's represent the weights associated with the connections going into the perceptrons (and the weight bias for the perceptron), and $g(net)$ represents the output of the perceptron. Thus $g\prime(net)$ is given by:

$$g\prime(net) = (g(net) - a_{min})(1 - \frac{1}{a_{max} - a_{min}}(g(net) - a_{min})) \tag{3}$$

To execute the backpropogation learning algorithm, the equations in (4.5) in Tom Mitchell's *Machine Learning* are used to iteratively update the weights in the network. Furthermore, the Δw_i's are updated using equation (4.18) in Tom Mitchell's *Machine Learning* to add momentum. The weights w and partial derivatives $\frac{\partial E}{\partial w}$ seen in (4.5) are calculated based on the model seen in slide 23 of '14-bp.pdf,' where E is the sum-squared error function being optimized.

The error function E being used is given by:

$$E = \frac{1}{2}\sum_p \sum_k (t_{pk} - y_{pk})^2 + \frac{1}{2}\gamma \sum w^2 \tag{4}$$

where p iterates over all the given training or test examples and k iterates over the dimensionality of the output vectors. The elements t_{pk} come from the elements of the target output vectors of the training and test data set, and the elements y_{pk} come from the elements of the output vectors produced by the neural network.The w's are added to implement a penalty term for the algorithm to seek small weight values (as seen in Tom Mitchell's *Machine Learning* section 4.8.1).

If there are hidden units in the neural network, there are a set of weights for the output layer and a set of weights for the hidden layer (as seen in slide 23 of '14-bp.pdf'). The partial derivatives $\frac{\partial E}{\partial w}$ for the output layer weights are given by:

$$-\frac{\partial E}{\partial w_{kj}} = -\sum_p \frac{\partial E}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial w_{kj}} - \gamma w_{kj} \tag{5}$$

$$= \sum_p (t_{pk} - y_{pk}) g\prime(net_{pk}) z_{pj} - \gamma w_{kj} \tag{6}$$

$$= \sum_p \delta_{pk} z_{pj} - \gamma w_{kj} \tag{7}$$

where $net_{pk} = \sum_j w_{kj} z_{pj}$. The partial derivatives for the input layer weights are given by:

$$
\begin{aligned}
-\frac{\partial E}{\partial w_{ji}} &= -\sum_p \frac{\partial E}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ji}} - \gamma w_{ji} && (8)\\
&= -\sum_p \frac{\partial E}{\partial z_{pj}} \frac{\partial z_{pj}}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ji}} - \gamma w_{ji} && (9)\\
&= -\sum_p (\sum_k \frac{\partial E}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial z_{pj}}) \frac{\partial z_{pj}}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ji}} - \gamma w_{ji} && (10)\\
&= -\sum_p (\sum_k (t_{pk} - y_{pk}) g\prime(net_{pk}) w_{kj}) g\prime(net_{pj}) x_{pi} - \gamma w_{ji} && (11)\\
&= -\sum_p (\sum_k \delta_{pk} w_{kj}) g\prime(net_{pj}) x_{pi} - \gamma w_{ji} && (12)\\
&= -\sum_p \delta_{pj} x_{pi} - \gamma w_{ji} && (13)
\end{aligned}
$$

If there are no hidden units, the partial derivatives $\frac{\partial E}{\partial w}$ for the output layer weights are similar to the equation in (6):

$$
-\frac{\partial E}{\partial w_{ki}} = \sum_p (t_{pk} - y_{pk}) g\prime(net_{pk}) x_{pi} - \gamma w_{ki} \quad (14)
$$

The given equations for the partial derivatives up to this point are to be implemented for batch learning. For online learning, p simply iterates over one example.

Experiments were run to determine what number of hidden units would produce the minimum error over the cancer and lenses testing data sets. Sample traces of the running program are given by 'lenses-trace.txt' and 'cancer-trace.txt.' The results of these runs are produced in 'lenses-results.log' and 'cancer-results.log.' The parameters used for the experiments are specified in 'cancer-bp.cfg' and 'lenses-bp.cfg.' The parameter d_{hid} varied from 0 to 10. The test data set sum-squared error (equation (4)) was graphed against the various hidden layer sizes for the cancer and lenses data sets (given in 'cancer-SSE.pdf' and 'lenses-SSE.pdf').

Based on the graph, it would be best to choose 1 hidden layer node for the lenses data set and 0 hidden layer nodes for the cancer data set. The optimal number of hidden layer nodes being low (for cancer and lenses data) could be a reflection of the possibility that the training data for the two data sets are close to linearly separable. This should be a higher number when dealing with a set of data that wouldn't be linearly separable.

I learned that even though neural networks can serve as a way to solve the classification problem by modelling any function, they can be computationally expensive. This is because the number of nodes in the network would drastically expanded the number of connections between then (and thus the computations for the various connections). Another problem that can arise is that the network can land on a local minimum instead of a global minimum (depending on the function being optimized) depending on how the learning rate is chosen.

Acknowledgements/References:

Mitchell, Tom M. *Machine Learning.* New York: Mc-Graw Hill, 1997. Print.

Noelle, David C. *Backpropogation of Error.* University of California, Merced, 2013. University Lecture.