Problem

The problem given is a standard classification problem. We are given a feature vector of size 16,562 with one output (either 0 or 1). There are 530 instances of such vectors, each having a class value of either 0 or 1.

There are many classification algorithms which solve this problem. But I choose Neural Networks because neural networks are scalable; they can easily be extended to a multi-class classification by adding neurons in the output layer; Accuracy can be increased by adding layers; easy to add biases; can be used a linear model as well as non-linear. Of course, there are easy to implement classifiers like the k-nearest neighbor or linear regression.

Model

Neural networks work well when the classification function is a continuous function and that we are we are good with a 92 % accuracy. I have implemented a "single unit neural network" modeled as below:

$$X_{V\times N}\ is\ the\ input\ vector\ with\ size\ V = 420, N = 16562,$$

$$W_{N\times 1}\ are\ weights, randomly\ assigned\ at\ first$$

$$U_{V\times 1} = X_{V\times N} \times\ W_{N\times 1}\ is\ a\ hidden\ layer$$

$$Y_{V\times 1} = sigmoid(\ U_{V\times 1}\ ) is\ the\ output\ vector\ (Predicted)$$

$$T_{V\times 1}\ is\ the\ actual\ output\ vector\ (Expected)$$

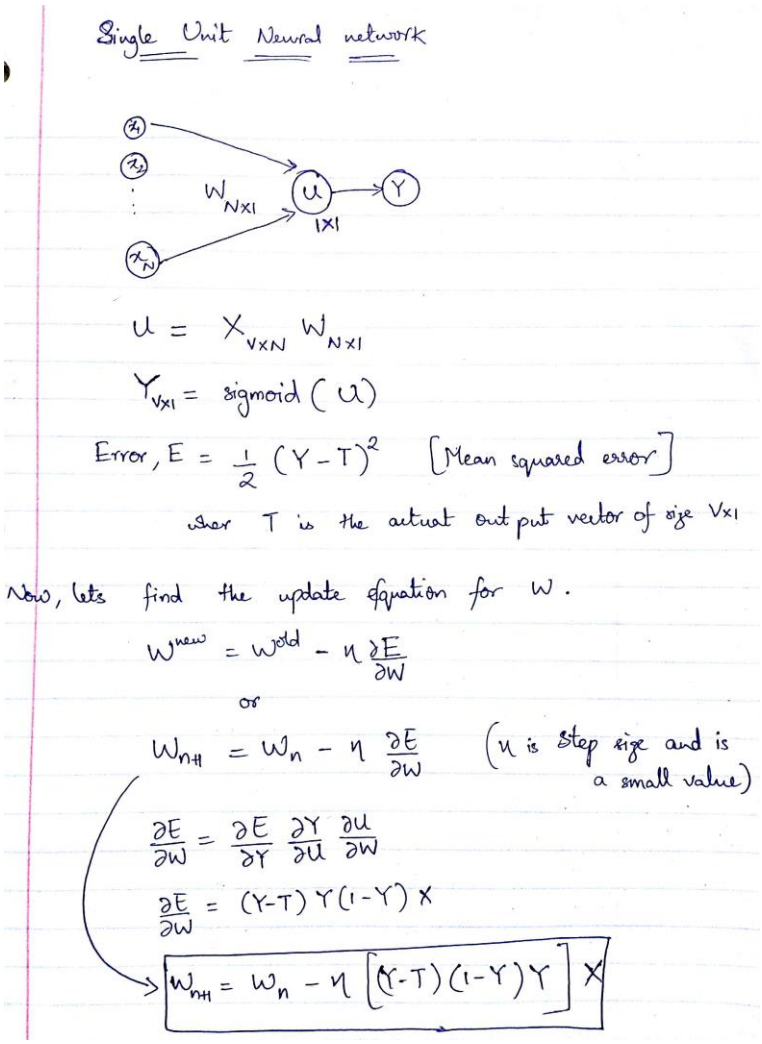
We define our error function as the mean squared difference between output vector Y and actual output T. Thus

$$E = \frac{1}{2}(Y - T)^2$$

$$E = \frac{1}{2}(\sigma(U) - T)^2$$

$$E(W) = \frac{1}{2}(\sigma(X\times W) - T)^2$$

Update equation is given in the next page (below the model) –



Code break though

The code initializes some constants in the beginning. I have divided 80% of the data into training data and the rest I am using as testing data. The program takes input in lines 69-83.

The heart of the algorithm is in the function *find_weights*, where we feed forward and back-propagate training data. Through feed forward we find the output value calculated by the neural networks. There is a for loop (lines 39-49) which runs 10000 times which updates weights as given by the update equation above.

Accuracy of the Model

The accuracy is calculated for the test data, given weights. There is a function *accuracy* which does the job. We assign value 0 if the output is less than 0.5, a value 1 otherwise. We see that the predicted output is close of 0 if the expected output is 0. And the predicted output is close to 1 if the expected is 1. The algorithm correctly predicted 107 values out of 110. So, the accuracy we got is 97.27.

There are ways to increase the accuracy. There are pros and cons of each method:

1. By increasing the hidden layers – we might overfit the data if we increase the power of our function, which leads to classifying the noise.
2. By adding a bias to the hidden layer – since there is no bias added in our model, our model will predict a 0 value when input X has all zeros. But the expected output might be 1. Adding is fairly easy to do (just a small change in the *feed_forward* function).
3. Also, by reducing the number of layers, we might underfit.
4. By decreasing the number of iterations:
5. What I've seen is that the difference between predicted and the expected values is more when iterations are less and vice versa. So, it really depends how accurate we want the results to be. Varying the iterations varies the time taken by the algorithm but not the complexity. Also, I've seen that for this data, and the way we are calculating the predicted output, 500 iterations are good enough.

Sl.no	# of iterations	Time taken to find weights	Accuracy
1	500	15 seconds	98.1%
2	10,000	300 seconds	97.2%