Laboratory 1: Neural Network

Overview

Neural networks are mathematical models based on our own biological neural structures. Considering this, a neural network is composed by multiple neurons (or nodes), where each one of them takes multiple inputs and calculates a single output. In order to compute the output, all nodes use a weight, which will be multiplied by the input values. The neuron will combine these weighted inputs, together with an activation function, in order to determine its output, simulating in this way the basic operation of how real neurons work.

In the matter of this assignment the objective will be finding the corresponding weights parameters (net-work calibration) using optimization techniques. A specific structure for this assignment is proposed: [4,2,1] (as it can be seen in Figure 1). Additionally, in order to bring closer the theory to the implementation, a different syntaxis is proposed for naming the nodes and weights.

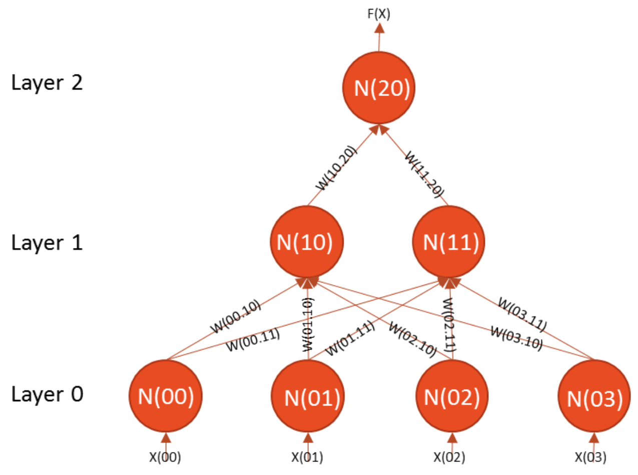


Figure 1 Proposed Neural Network

$$I_{ln} = f\left(\sum_{m=0}^{M}\sum_{n=0}^{N} W_{(l-1)m.ln} * O_{(l-1)m}\right)$$

Equation 2 Calculation of Each Node's Output

Within this configuration the input of each one of the nodes is calculated by summing the multiplication of each one of the previous outputs by the corresponding weight of the incoming connection and passing that value through the activation function, which in this case is the Sigmoid Function (as it can be seen in Equation 1).

$$s(x) = \frac{1}{1+e^{-x}} \qquad s'(x) = s(x)\cdot(1-s(x))$$

Equation 1 Sigmoid Function and its derivate

As stated, the inputs of each node can be understood as presented in Equation 2, where l is the layer of the node, n is the corresponding number of the node in the layer and m is the number of nodes in the previous layer.

Considering this, we can finally have that for the proposed structure the formula that calculates the output of the end node, presented in Equation 3 and Equation 4.

$$O_{20} = f(I_{20}) = f(W_{10.20} * O_{10} + W_{11.20} * O_{11})$$

Equation 3 Final Output Calculation

$$O_{20} = f\left(W_{10.20} * f\left(\sum_{m=1}^{4} W_{1m.10} * f(X_{0m})\right) + W_{11.20} * f\left(\sum_{m=1}^{4} W_{1m.11} * f(X_{0m})\right)\right)$$

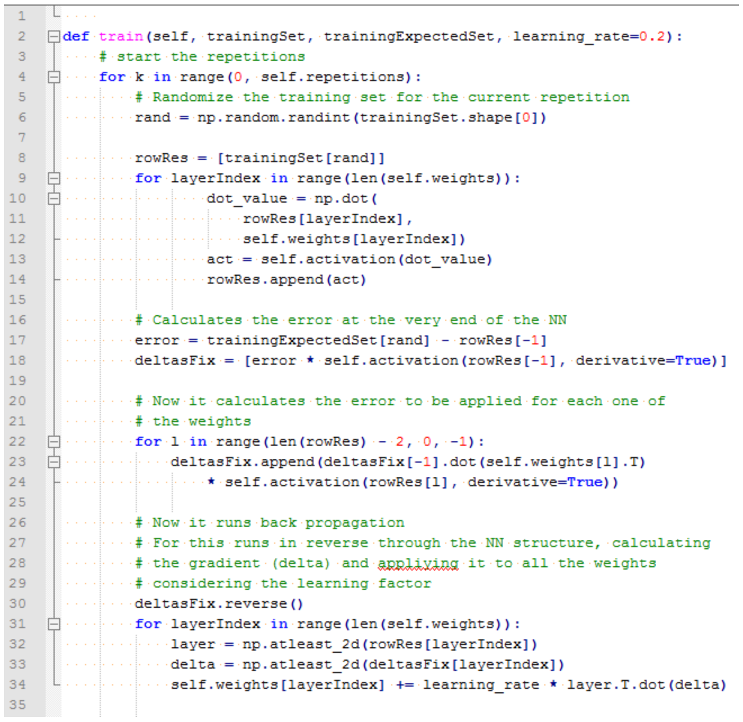
Equation 4 Final Output Calculation Detailed

An interesting conclusion that can be derived from the inner nature of the assignment is that it is in fact a non-convex optimization problem. This can be concluded based in the dependency to the sigmoid function, as the activation for the Neural Network, which is not-convex. Based on that the solutions proposed for this assignment are not based in algorithms or solvers for convex functions.

Solution

For the current assignment, and after researching different approaches to resolve the non-convex problem presented, two different solutions have been developed, in this case in Python, that can give a better understanding of how a Neural Network works and how can a problem of this nature be approached. It is also important to highlight that even though we have a proposed structure for the neural network, the solution developed was made to be configurable so it can receive different structures and find the optimal weights (please refer to Annex 1).

The first solution is a local development of the Backpropagation Algorithm in order to find the most optimal weights corresponding to the structure of the neural network. Using this algorithm, which in the current implementation didn't include the implementation of biases, an error is determined after running the inputs to the network and a portion of it is propagated backward (hence the name of the algorithm). At each neuron in the network the error is used to adjust the weights and threshold values of the neuron, so that the next time, the error in the network response will be less for the same inputs[1].



```
1
2  def train(self, trainingSet, trainingExpectedSet, learning_rate=0.2):
3      # start the repetitions
4      for k in range(0, self.repetitions):
5          # Randomize the training set for the current repetition
6          rand = np.random.randint(trainingSet.shape[0])
7
8          rowRes = [trainingSet[rand]]
9          for layerIndex in range(len(self.weights)):
10                 dot_value = np.dot(
11                     rowRes[layerIndex],
12                     self.weights[layerIndex])
13                 act = self.activation(dot_value)
14                 rowRes.append(act)
15
16         # Calculates the error at the very end of the NN
17         error = trainingExpectedSet[rand] - rowRes[-1]
18         deltasFix = [error * self.activation(rowRes[-1], derivative=True)]
19
20         # Now it calculates the error to be applied for each one of
21         # the weights
22         for l in range(len(rowRes) - 2, 0, -1):
23             deltasFix.append(deltasFix[-1].dot(self.weights[l].T)
24                 * self.activation(rowRes[l], derivative=True))
25
26         # Now it runs back propagation
27         # For this runs in reverse through the NN structure, calculating
28         # the gradient (delta) and appliying it to all the weights
29         # considering the learning factor
30         deltasFix.reverse()
31         for layerIndex in range(len(self.weights)):
32             layer = np.atleast_2d(rowRes[layerIndex])
33             delta = np.atleast_2d(deltasFix[layerIndex])
34             self.weights[layerIndex] += learning_rate * layer.T.dot(delta)
35
```

Figure 2 Backpropagation Learning Implementation in Python

The implementation of this learning can be seen in the extract of code of Figure 2, where the training is run by a static number of repetitions, *10000* in our case, and for each repetition the output of the network based on the complete input set is calculated (in lines 8 to 14 of Figure 2) so the weight adjustment (learning) can be processed as explained in the equations below.

$$W_{(l-1)m.ln} = W'_{(l-1)m.ln} + LR \cdot e_{ln} \cdot X_{ln}$$

Equation 5 Calculation of the weights

$$e_{ln} = s'(Y_{ln}) \cdot (d_{ln} - Y_{ln})$$

Equation 6 Error calculation for output layer

$$e_{ln} = s'(Y_{ln}) \cdot \sum (e_{(l+1)n} \cdot W'_{(l-1)m.ln})$$

Equation 7 Error calculation for internal layers

Here we can observe that each weight is calculated using the general formula seen in Equation 5, where $W'_{(l-1)m.ln}$ is the previous value of the weight, LR is the current learning rate that is being used by the algorithm (in our implementation we have a constant value of *0.2*), e_{ln} is an error term calculated either by Equation 6 or Equation 7, and X_{ln} is the input of each neuron.

In order to calculate the error term, two cases are considered: the first case is for the output layer, where the error term is the multiplication between the actual output Y_{ln}, passed through the derivate of the sigmoid function (as seen in Equation 1), and the difference between the desired output d_{ln} and the actual output (which can be seen in line codes 17 and 18 of Figure 2). Meanwhile, for the case of the internal layers, the difference between the desired output and the actual output is replaced by the sum of the error terms for each neuron in the layer immediately succeeding the layer being $e_{(l+1)n}$ times the respective pre-adjustment weights $W'_{(l-1)m.ln}$ (which can be seen in lines 31 to 34 of Figure 2).

The second solution developed is based on the Truncated Newton Algorithm, also called Newton Conjugate-Gradient. The algorithm incorporates the bound constraints by determining the descent direction as in an unconstrained truncated Newton, but never taking a step-size large enough to leave the space of feasible X's. The algorithm keeps track of a set of currently active

[1] Cheshire Engineering Corporation. (2003). NeuralystTM User's Guide. Retrieved from http://www.cheshireeng.com/Neuralyst/nnbg.htm

constraints, and ignores them when computing the minimum allowable step size. (The X's associated with the active constraint are kept fixed.)[2]

In our case we make use of FFNet[3], a Python library that relies on ScyPy[4] by using the *fmin_tnc* optimizer to find the respective weights of a Neural Network. Apart from the solving algorithm used, this solution differs from the first solution by including the use of biases to determine the most optimal weights for the network.

Results

Four testing sets were used to test the implemented solutions, and for each set, three runs were performed in order to have a more accurate understanding of its performance. It is important to highlight that the solution implemented takes 80% of the input records to train the neural network and leaves the 20% left to test it. The numeric results of the performance test can be seen in the Table 1 below.

Solution:	**Backpropagation**								**Trucated Newton**							
Size of Input Set	16 records		128 records		1024 records		16384 records		16 records		128 records		1024 records		16384 records	
	Accuracy %	Duration (s)	Accuracy %	Duration (s)	Accuracy %	Duration (s)	Accuracy %	Duration (s)	Accuracy %	Duration (s)	Accuracy %	Duration (s)	Accuracy %	Duration (s)	Accuracy %	Duration (s)
Run 1	50.00	5.44	69.23	5.31	73.65	4.96	76.17	5.08	100.00	0.17	100.00	0.30	100.00	2.70	99.93	27.77
Run 2	50.00	5.86	57.69	5.45	77.25	5.63	70.74	5.95	100.00	0.24	100.00	0.33	100.00	2.82	99.96	24.95
Run 3	75.00	5.72	69.23	5.99	78.53	5.59	84.22	5.59	100.00	0.18	100.00	0.30	100.00	2.71	99.97	19.08
Average	**58.33**	**5.67**	**65.38**	**5.59**	**76.48**	**5.39**	**77.04**	**5.54**	**100.00**	**0.20**	**100.00**	**0.31**	**100.00**	**2.74**	**99.95**	**23.93**

Table 1 Performance Results

Final Comments

Considering the results obtained by testing both solutions it is clear that the implementation that uses the Truncated Newton algorithm reaches an outstanding accuracy for the problem to be solved. But apart from that there are other aspects that are interesting to highlight taking into account the results gathered.

In terms of performance, it is interesting to see that the time the second solution takes to run depends directly on the number of records in the input set, meanwhile for the first solution, whose accuracy seems to increment with bigger inputs, the duration remains almost invariable even though the input set increments drastically.

Similarly we can appreciate that with larger inputs the second solution start presenting lower accuracy that wasn't shown in previous iterations, indicating that, even though is a good solution for the presented problem, the algorithm does not find the optimal solution, but it stops when it considers that the solution is good enough, due this the name 'truncated'.

Finally, we can conclude that, even though backpropagation is a good method to approach this problem, in order to improve accuracy in the first solution, more alternatives and further work has to be implemented (as including biases, optimizing the learning rate, among others).

If necessary, attached to this report you will find a technical how-to-use guide for the solution implemented, with all the detailed instructions and requirements for using it.

[2] The Scipy community. (2014, May). scipy.optimize.fmin_tnc. Retrieved from Optimization and root finding (scipy.optimize): http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.fmin_tnc.html

[3] Feed-forward neural network for python (ffnet). http://ffnet.sourceforge.net/

[4] SciPy. http://scipy.org/